MASTER'S THESIS

DETECTING THE PRES ENCE OF BOTNETS BY ANALYSING TCP/IP NETWORK TRAFFIC

van Renswou, J.M.

Award date:
2021

Open Universiteit
www.ou.nl

DETECTING THE PRESENCE OF BOTNETS BY ANALYSING TCP/IP NETWORK TRAFFIC

Student: ing. J.M. van Renswou BBA
Student number:
Date of presentation: February 2nd, 2021

Open Universiteit
www.ou.nl

DETECTING THE PRESENCE OF BOTNETS BY ANALYSING TCP/IP NETWORK TRAFFIC

A master thesis submitted by

ing. J.M. v Renswou BBA

Open University of the Netherlands,
Faculty of Science

Master's Programme in Software Engineering

Student:	ing. J.M. van Renswou BBA
Student number:	851932446
Date of presentation:	February 2nd, 2021
Course:	IM9906
Degree programme:	Open University of the Netherlands, Faculty of Science Master's Programme in Software Engineering
Chairman:	dr. ir. H. Vranken
Supervisor:	dhr. dr. A. Hommersom
Thesis committee:	dr. ir. H. Vranken, dhr. dr. A. Hommersom

Contents

Abstract

The almost unlimited opportunities of the internet are not always a positive thing. Not everyone is using the internet for the good. Some are using these opportunities for activities that are unwanted. One example are botnets that consist of networks of highjacked computers to do criminal actions. This is a major thread that affects everyone connected to internet.

In this research an attempt is made to detect the computers that are highjacked to be part of a botnet by monitoring their network behaviour.
The main research question is:

How can machine learning techniques effectively and efficiently detect botnets from TCP/IP network traffic?

By selecting three machine learning techniques, models are created using publicly available datasets with network traffic coming from botnets and normal programs. The used machine learning techniques are Random Forest Classifier (RFC) Support Vector Machines (SVM) and Gradient boosted Trees (GBT). Using a flow-based approach with only 17 features per flow an RFC could be trained to detect the botnet network traffic. With an accuracy of 99.63%, it is performing better than the SVM Classifier and the GBT Classifier, on the validation data. The small number of features used assures a low algorithm complexity. A low complex algorithm will reduce the change for overfitting and reduce the resources needed to evaluate a new flow.

With network flows that contain network packets from new botnets and normal program traffic, the RFC performance is poor, with an accuracy of only 55.59%. The features extracted from the flows are good to detect known botnets but are not generic enough to distinguish network flows from unknown botnets and normal programs.

To train an algorithm, multiple datasets are available. Only not all datasets use the same method for adding a truth label to the network packets. Therefore, a software package is created to convert different datasets to flows and add a truth value to each flow.

The complete software package, called **botshot**, can be used to convert datasets to flows, create feature-sets from flows, train a machine learning model from the feature-sets and validate the performance of the used machine learning algorithm. The software package is documented, and the architecture makes it easy to be adapted to new datasets, export different features or try new training algorithms. The **botshot** software package will help new research to focus on selecting new features and better classification algorithms instead of spending time in converting raw data to usable features and test the performance.

1. Introduction

The internet is giving us almost unlimited opportunities to explore a new parallel digital universe. Practically everyone is part of this new universe and participates one way or another. People have a digital life and companies have new ways to communicate to consumers. The internet is still evolving and changing. It is a part of our daily lives.

Not only the good citizens are using the internet, but also the ones that are not playing by the rules. Just as in the real world, there are threats out there where you should look out for. In the real world, there are government organizations that protect good citizens as good as possible. Years of experience and a clear environment helps these organizations create a safe surrounding for most citizens.

In the digital world, this is different. The environment is still changing rapidly and there are no country borders. The protection of citizens is becoming difficult. The criminal actions in the digital world are different than in the real world. Computer geniuses and script kiddies are using their computer skills to earn money with illegal activities.

One of those methods to earn money illegally is with botnets (Khattak et al. 2014), a combination of the word's "robot" and "network". Botnets are a group of computers that are highjacked by a single person or group to do criminal actions. On their behalf, the group of computers can perform actions, hiding the owner or controller of the botnet. A bot that is part of a botnet can be an ordinary personal computer connected to the internet or a smart camera in a home. Every device connected to the internet can be part of a botnet. Mostly the owner of the device is unaware that it is part of a botnet (Nadji et al. 2013).

A botnet can be used for all kinds of illegal activities (Zeidanloo et al. 2010). Examples are, the sending of spam email or to create a distributed denial of service (ddos) to overload a server connected to the internet. Botnets can also be used to steal data or spread ransomware. Not only large companies can be a victim of botnets, but also a single person can be targeted, for example, with identity fraud.

Detecting botnets is difficult (Alauthman et al. 2020). It can be done on the computer by a detection program. This works for a personal computer or server, but it is more difficult for smaller devices connected to the internet, like smart cameras. It is also challenging to enforce every device on the internet to use protection software. Another option is to search actively for the person or persons who are controlling the botnet. This requires that a lot of organizations be working together. There are examples that this can be successful but is challenging to organize (Nadji et al. 2013). Another option is to isolate the infected device, partly from the internet, only stopping the network traffic from the botnet program.

To isolate an infected device from the internet, it should first be identified as being part of a botnet. The detection can be done by inspecting the behaviour of the device on the internet. At, for example, internet service providers or internet exchange points, the network traffic from devices can be monitored and evaluated if the device is part of a botnet. When a positive detection is done, the device can, be isolated from the rest of the internet or the network traffic that is generated by the botnet can be stopped. Also, the exact behaviour of the botnet can be studied and maybe it is possible to find the owner of the botnet and bring him to justice.

A vital remark to a positive detection is a false positive detection. When an action is taken it should be clear that the device is infected. Automated detection can make mistakes, and this should be taken into consideration when action is taken.

The purpose of this research is to evaluate the possibilities to detect botnets on devices by their behaviour on the internet. Network packets are inspected and processed by a machine learning algorithm. A considerable amount of research is done in this area. Especially in the last years, the detection of botnets with deep neural networks is researched intensively. In our research, we want to use a more classical machine learning algorithm, because the training and execution of these algorithms are mostly much faster (Widanapathirana et al. 2012) and will help to process a larger amount of network traffic with less computer power. We will not only test the effectiveness of our trained classifier with the data we prepared but also try to test our classifier against a data set with network traffic that is not used during training, this to simulate a more real-world situation.

In this chapter, the related work and research questions are presented. In chapter 2, some background about the used technology is discussed. Chapter 3 is about the methods used to answer the research questions. And the used data to train a machine-learning algorithm is discussed in chapter 4. To transform the data in a usable format, a program is created. The working of this program is discussed in chapter 5. Also, in chapter 5, a program is discussed to train different machine learning algorithms. Chapter 6 presents all the done experiments and chapter 7 concludes this thesis.

1.1. Related work

Botnets are an increasing problem on the internet. To have some understanding of what botnets are and how they operate, Khattak et al. (Khattak et al. 2014) have made an excellent overview. The different incarnations of botnets are described and how they are controlled. An explanation is given on the methods botnet use to hide their controller, the botmaster, and how the botnets behave. The terminology used for botnets is summarized and used in this report. The detection of botnets through their behaviour is an active research area. In our case, we focus on the detection of botnets through their behaviour on the network (internet).

Prior studies in this area have achieved excellent results, and accuracy results get close to 100%. Roosmalen et al. (Roosmalen et al. 2018) reported a close to 100% accuracy of detecting network traffic coming from Botnets. This excellent result was achieved using a vast neural network. Due to the considerable dataset and vast neural network, the resources needed to train and evaluate this neural network are significant. van Klaveren (van Klaveren 2019) used this work as a basis for his thesis to explore the possibilities to reduce the number of features needed to train a neural network for detecting Botnet network traffic. With different approaches, he tried to extract knowledge from the neural network to predict which feature have a high importance for the neural network. Because it was difficult to predict which features are important for the neural network, he tried a more empirical approach to see which features have a higher importance for the network. By removing features and evaluate the accuracy of the new created model. The drop in accuracy is a measure of how important the feature is for the neural network to have an accurate prediction.

Wang et al. (Wang et al. 2020) introduced Botmark with an almost perfect detection performance. Botmark is a combination of flow-based and graph-based detectors. The created graphs map the communication patterns that exists in the datasets, combining multiple flows. The detection algorithms are using small features sets and use whitelists to reduce the network traffic to be inspected. With these techniques, the proposed solution can be used relatively quickly because it reduces the resources needed for training and evaluation. Also, Pektaş et al. (Pektaş et al. 2019) use multiple techniques to detect botnets in network traffic. By analysing network traffic and calculating features from the network traffic flow, a novel combination of types of neural networks can be used to detect the botnets. The novelty is to use a combination of CNN, LSTM, and fully connected neural networks to reduce the overall network size and increase the accuracy.

There are a lot of machine learning techniques available in the literature. Every method has its specific areas where it can be applied best. Fernandez-Delgado et al. (Fernandez-Delgado et al. 2014) have created an overview of 179 classical classifiers and tested them on 121 different datasets. On the tested datasets, the Random Forest, (Ho 1995) and SVM (Cortes and Vapnik 1995) have the best accuracy from all tested classifiers. Examples where SVM and random forest are used for detecting botnets are Lin et al. (Lin et al. 2014) and Chen et al. (Chen et al. 2017). They are both a little bit older because recent research is almost always done with a neural network. In a paper from Alauthman et al. (Alauthman et al. 2020) a Gaussian Mixture Models (GMM) is presented to classify network traffic. The model is compared against 6 different other classifiers. Outside the GMM classifier, the Gradient Boosted Trees (GBT) (Friedman 2001) classifier is also performing well on network traffic.

For most machine learning techniques, it is necessary to have data to train and evaluate the generated models. For botnets, there is a selection of datasets available that can be used to do the training: UNB Botnet (Biglar Beigi et al. 2014), Bot-IoT (Koroniotis et al. 2019), CTU-13 (García et al. 2014). These datasets are large enough to train and validate most machine learning algorithms. The datasets consist of network packets from traffic between client and server. The most used protocol between clients and servers on the internet is TCP/IP (rfc 793 1981). To have a better understanding of the TCP/IP network protocol and other network protocols used on the internet the book "The TCP/IP guide: A comprehensive, illustrated internet protocols reference (Kozierok 2005) can be used. The other protocol that is used often in networks is UPD/IP (rfc 793 1981). Bots in a botnet will mostly use one of these protocols to communicate with each other.

From every dataset, features are selected or calculated to be used for training a machine learning algorithm. For example, Roosmalen (Roosmalen et al. 2018) have extracted network flows from the individual network packets available in the dataset and create features from these flows. A network flow is a set of network packets that belong to each other.

Which features to select can be a difficult task. It is not always trivial to generate features from a dataset that are meaningful. Guyon et al. (Guyon et al. 2003) discuss the problem of variable (feature) selection and present a helpful checklist to help to select the best features for the challenge.

Chandrashekar et al. (Chandrashekar et al. 2014) give an overview of techniques on how to selects important features for machine learning. But as van Klaveren (van Klaveren 2019) has shown it is not always possible to reduce the

number of features automatically. Algorithms are not always giving a good result and the computational cost trying all combinations is high.
Training a machine learning algorithm should be done with a generalized dataset. A machine learning model should give valid results when new data is evaluated. In more research areas, the question is asked, how the models can be extrapolated to cover the complete domain. In a paper from Moons et al. (Moons et al. 2009), a beautiful example is given about this problem for clinical trials. The predictions problem encountered with diagnostic models in primary and secondary patients can be translated to botnet detection. Will the network traffic from a new botnet, that is not used for training, be detected by an algorithm, trained with network traffic from other botnets? Most research about botnets is not trying to validate their algorithm with new, unseen data from new Botnets.

The usage of accuracy (Sokolova et al. 2009) is quite common to express the performance of machine learning algorithms. The disadvantage of using the accuracy is that it is challenging to compare performances when there are different datasets used to evaluate the machine learning algorithms. When the ratio of positive and negative data is different in the used dataset, the accuracy number is not a good number to compare the performance of an algorithm. Sokolova et al. (Sokolova et al. 2009) introduces more performance measures for classification tasks that are often used to give the performance of a machine-learning algorithm. These parameters have the same problems with biased datasets as accuracy. Machine learning algorithms tend to predict the most abundant class. A Matthews Correlation Coefficient (MCC) (Matthews 1975) is proposed as an alternative for accuracy, to overcome the problem with performance measures for imbalanced datasets. Another method of analysing the performance of a classifier is the use of Receiver operating characteristics (ROC) graphs. In "An introduction to ROC analysis" Fawcett (Fawcett 2006) gives an introduction to ROC. This can be used as a guide on how to read the generated graphs.

1.2. Research

Knowing that a personal computer or more general, a device that connects to the internet, is part of a larger botnet is essential. The usage of Botnets is almost always for criminal actions. The owner of the botnet (Botmaster) will use the botnet to get an advantage (Khattak et al. 2014). In an annual report from the FBI (Gorham, Matt 2019) the total losses due to internet crime is exceeding $3.5 Billion in 2019, showing the need for better protection and detecting of criminal internet activity.

Many techniques are available to prevent devices becoming part of a botnet or to detect the botnet software that runs on the device (Drašar et al. 2014). Additionally, it can be important to detect devices that are part of a botnet through their behaviour. Because devices belonging to a botnet have almost always a connection to the internet, at some point in time, the behaviour on the internet can be an indication that the device is part of a botnet. A lot of research has been done in detecting botnets through their behaviour on the network. Not all proposed methods are using features that are independent of the network topology. Some used features are so specific for the training data, that it is questionable if the results can be reproduced with a newly created dataset for validation. The results are so specific for the current situation that detection will fail if the network environment is changed. A created algorithm and used features should be independent of the network topology (Nadji et al. 2013).

An algorithm should be able to classify new network traffic without huge mistakes.

The results from most recent research are obtained using deep neural networks. Although this research has promising outcomes, the resources needed to train and run the deep neural networks are considerable (Widanapathirana et al. 2012). The resources needed to train a new model and evaluate new network traffic is important. The amount of network traffic to evaluate can be very large and the reaction time to create new models should be fast to minimize the losses. It should be possible to compare algorithms on the usage of resources. The practical use of an algorithm is dependent on the availability of the resources.

1.2.1. Questions

From the wish to detect botnets from network behaviour, we can define the following research question.

How can machine learning techniques effectively and efficiently detect botnets from TCP/IP network traffic?

In the research question, the used technique to detect botnets will be learning-based. The opposite of learning-based is rule-based. Rule-based algorithms are difficult to make general. The differences in network traffic from different botnets make it difficult to find detection rules. By using machine learning techniques, it is possible to evaluate a large amount of data to create a detection model. Not only how effective the algorithm is in detecting botnets is important but also how efficient it can do this. When the amount of network traffic is increasing the efficiency will be important (Vasques et al. 2019).
From the research question, several sub-questions can be defined.

RQ1 Which dataset will be used to train and evaluate machine learning algorithms?

Machine learning uses examples of data belonging to one of the classes that need to be detected. In the learning phase of the machine learning algorithm, data is presented to the algorithm with a truth label. The truth label classifies if a network packet belongs to the bot software running on a computer being part of a botnet or is coming from regular network traffic. Having a useful and large enough dataset, which is of good quality, is important for every training algorithm to get the best results (Cortes, Jackel, et al. 1995).

RQ2 Which training features, from TCP/IP network traffic, should be used for botnet detection?

A dataset is a set of raw data with a truth value. To be successful, this data should be pre-processed before it can be used for training a machine learning algorithm. There are multiple options to gather extra information from successive TCP/IP network packets. Also, not all data in the TCP/IP network packets have a meaning. Encrypted data inside the package is difficult to read, but the size of the data can be useful. Getting the right features is probably more important than the used machine learning algorithm (Guyon et al. 2003).

RQ3 What selection of machine learning algorithms can be used best to detect botnets?

A large amount of machine learning algorithms is available, which all have their specific usage. Because of the nature of the provided problem and the size of the dataset used for training, not every machine learning algorithm is suitable for this problem. A shortlist of algorithms must be made to try on the selected dataset.

RQ4 How can the different algorithms be compared in their effectiveness and efficiency in detecting botnets?

The effectiveness of an algorithm depends on different areas. During the training of a machine-learning algorithm, the dataset is normally split in a set for training and a set for validation. The performance of the trained algorithm on the validation set can be a good measure of the effectiveness/performance of an algorithm (Wong 2015). Another measure can be the ability for detecting botnets in a dataset that is not used during training. Ideally, the new dataset has samples of botnet network traffic that was not included in the dataset used for training. At last, the resource usage of the algorithm can be an important performance measure. Depending on the device where the algorithm is executed, the resources like time spent in evaluation or memory usage for training can be important.

Depending on the chosen method for evaluating the effectiveness of the used algorithms, the results can be compared with the results from other papers. Choosing a method that can be used for comparison is important to rate the results from this research. Only after comparison the results can be interpreted in the right context.

In chapter 4 the used datasets will be introduced for answering RQ1. The features used for training are explained in chapter 6, this will answer question RQ 2. The question RQ 3 will be answered in chapter 6 but the workings of the used algorithms are discussed in chapter 2.2. The last question RQ 4 will also be answered in chapter 6.5 experiment 4. A summary of all answers is given in the conclusion, chapter 7.

2. Background

In this chapter, the different technologies, that are used, will be discussed. This chapter is introducing the basics of the technologies and not an in-depth discussion about these technologies. This chapter is to get the basic knowledge about the technologies. When a basic understanding of the technology is present, this chapter can be skipped.

2.1. Botnets

Already some introduction to botnets was given in chapter 1. The information below is a summary of the paper from Khattak et al. (Khattak et al. 2014).

On average botnets are no good news and are mostly used for criminal actions. With a botnet a criminal tries to take over a computer. The owner of that computer is not aware his computer is under control of someone else. A computer that is part of a botnet can do actions on behalf of the controller of the botnet, the botmaster. The botmaster will try to hide his presence on the computer and when his presence is detected, hide his identity. The botnet computers are getting their command from one or multiple command and control servers. The command-and-control servers are directly controlled by the botmaster or by another Command-and-Control server acting as a proxy. Most botmasters are using more than one proxy to make it more difficult to find his identity. The proxy in between the botmaster and command and control servers are called steppingstones. In Figure 1, a typical structure of a botnet is given.

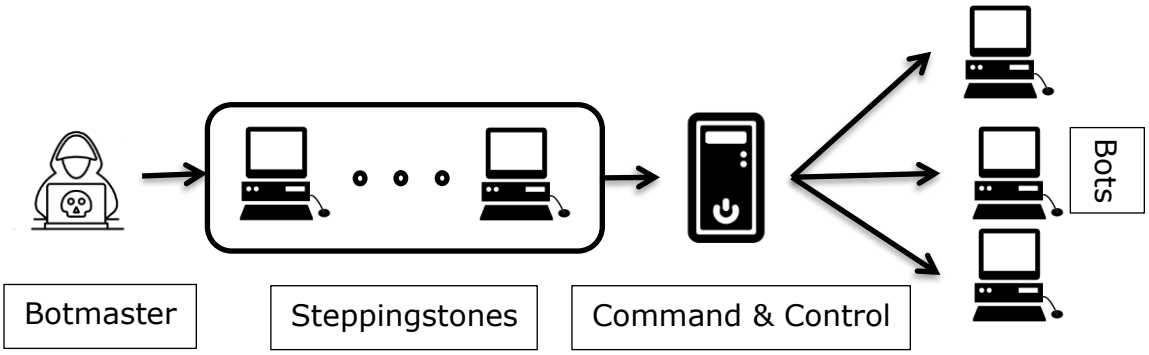


Figure 1: Structure of a Botnet

The first goal of a botnet is to propagate to increase the number of bots in the network. There are a couple of methods a botnet can use to infect new computers with the botnet software. The first method is called the active method. The botnet software is actively busy with recruiting new computers. The botnet software is looking for security problems to spread itself to more computers. In this method, there is no user intervention needed. In a second method, a botnet can use to spread itself is the passive mode. This method requires some sort of user intervention to infect the computer. The intervention can be visiting a particular prepared web site or the use of infected media (e.g., USB drives). Also, social engineering is used to let the user willingly download the botnet software on his computer.

When a computer is infected with the botnet software, it will try to contact a Command-and-Control server. The Command-and-Control server is a server connected through the internet. Getting the correct IP address of the Command-and-Control server can be done through DNS name resolving or static IP. The

botnet can also have a list of server names or addresses he can try. The botnet software will have multiple options to contact a Command-and-Control server even when authorities have taken some down. The botmaster will try to make sure a bot can always contact a Command-and-Control server. When a bot cannot reach a Command-and-Control server, the bot is useless.

The communication between the Command-and-Control server and the bot can be done with an existing communication protocol or with a proprietary protocol. When using dedicated network ports, a bot can be easily blocked with a firewall. Therefore, bots are increasable using HTTP to hide their communication. This makes it difficult to stop the communication between a Command-and-Control server and the bot with standard tools.

2.1.1. Purpose of a Botnet

The purpose of a botnet is to carry out malicious activities on behalf of the botmaster. Those malicious activities are done to earn money. An obvious activity is the sending of spam mail. With a large botnet the botmaster can send a huge amount of mail that is difficult to stop because of the distributed nature of the botnet. The mails can contain an advertisement to buy some goods or tries to provoke you to click on a link. The link can lead to a malicious website that infects your computer with malware. Another type of mail looks like it is coming from an official instance that asks you to enter some private information that can be used to get money from you. There are many types of malicious email types that can be used to earn money. The botmaster can send this email to have a direct benefit or can send this email on behalf of another criminal and get paid for it.

With a large botnet it is also possible to attack a specific webserver with a DDOS attack. With a DDOS attack, a huge amount of traffic is sent to a webserver. The webserver is unable handle the amount of network traffic and will be unavailable for other users. A DDOS attack can be done to make a political statement or to get a competitive advantage. It is also possible that a DDOS attack is used to hide other illegal activities like hacking the network of a company. As with spam mail botnets are most likely hired from the botmaster to do the DDOS attack.

The bots in a botnet can also be used to spy on the user of the computer. When the botmaster has control over the computer it can mostly access all the resources on that computer, including private information stored on that computer. The botmaster can record what the camera is seeing or listen with the microphone of the computer to the sounds in the room where the computer is located. The private information can be used to steal your identity to buy expensive goods or do a criminal activity on your account. The personal data can also be used to steal bank information and directly transfer your money to another bank account. This are only some examples of what a criminal can do with your private information. With the information from your computer, it is also possible to blackmail you directly. For example, with the images on the computer or taken with the camera the botmaster can threaten to expose those images to the world. It is not a pleasant idea that someone is looking around in your computer without you knowing it.

A last example of what a botmaster can do with his botnet is to take over the computers and servers from a company. When the botmaster has enough computers and servers infected inside a company, he can disable that network

and ransom the stored data on that network. There are examples enough in the media that shows that this is an enormous threat.

2.1.2. The topology of a Botnet

A botnet can use different topologies. The various topologies are used to spread the risk of detection and takedown. The botmaster wants to protect the botnet from instances that want to take down the botnet.

The first type of botnets has the structure, as presented in Figure 1. There is one Command and Control server that is reachable by the botmaster through the steppingstones. The Command-and-Control server is controlling the botnets. But this structure is sensitive to disruption. If one of the steppingstones or Command-and-Control server is disabled, the entire botnet is decommissioned.
Numerous architectures have been invented to hide the detection of the botmaster and make the botnet robust. Khattak et al. (Khattak et al. 2014) specified three different types of botnets:

1. Centralized
2. Decentralized
3. Hybrid

The Centralized botnet is the one described before. It has a single point of failure, making it sensitive. The advantage of a centralized botnet is the speed of reaction. Because the botmaster is closely connected to the bots, the bots can react quickly. This can be an advantage when a network of a company is infected, and the reaction of the botnet should be fast to counter the defence activities of the company. With a centralized botnet there two subcategories defined. The "star" category, where the botmaster is directly connected to the Command-and-Control server, or the hierarchical category where there are multiple proxies between the botmaster and the Command-and-Control server. As can be imagined, the closer the botmaster is connected to the Command-and-Control server the faster the botnet can react but the easier it is to expose the identity of the botmaster.

The Decentralized botnet is a reaction on the single point of failure of the Centralized botnet. In a Decentralized botnet, there is no single entity that is controlling the complete network. In the distributed subcategory the bots are communicating with multiple Command-and-Control servers and the Command-and-Control servers are communicating with the botmaster through multiple steppingstone paths. Because there are multiple paths from the botmaster to a bot, there is no single point of failure. Besides the advantage of no single point of failure, a distributed botnet can do load balancing and has better availability and resilience. Although a distributed botnet has advantages, it is more challenging to create. The software is more complex, and you need more computers that are part of the botnet. A second subcategory of decentralized botnets is "random" botnets. Any computer in a random botnet can be a bot, a Command-and-Control server, or a steppingstone. The paths from botmaster to bots are random, and undefined. Because information is reaching the bot through different paths, it is not easy to track. This makes it easier to hide the identity of the botmaster. There is no single point of failure; removing one computer from the botnet is not affecting the botnet at all. Because every bot can fulfil all functions, this botnet is difficult to disturb. The downside of this type of

decentralized bot is that it is difficult to have a coordinated attack. There are unpredictable delays in the network which makes coordination difficult.

Every botnet topology has its use. Depending on the actions a botmaster wants to do a topology can be used. More advanced topologies are proposed in the literature to overcome the different downsides discussed. It is a race between the good and the bad where technology is used to get an advantage.

2.1.3. Hiding the Botnet

An essential property of a botnet is that it operates stealthily. It should try to hide its activity on the infected computer, avoiding detection. Not only the bots operate stealthily but every step going back to the botmaster should be taken carefully to hide the botnet and the botmaster. For the bots and Command-and-Control server, there are different tactics to hide their presence.

A botnet tries to infect bots by exploiting known and unknown security vulnerabilities on the host computers. To avoid detection, the botnet is using technologies to hide his presence and detection by anti-malware software. Polymorphisms, disabling the anti-malware software and obfuscation are known technologies to avoid detection.

When it is known that a computer is infected with a botnet, it can be investigated to try to find the Command-and-Control server and eventually the botmaster. The software running on bots is trying to prevent this research. By detecting the environment, it runs on, it will try to see if it should stop functioning. For example, it is a known practice to run a copy of an infected computer in a virtual environment so it can be investigated. By detecting that the computer is running in a virtual environment, the botnet software can stop doing any action. This makes studying the software more difficult.

The botnet is trying to avoid being detected on the infected computer. It hides his presents on the computer from detection software and tries to hide the communication between the different botnet nodes. There is a long list of techniques to prevent capturing the IP traffic between the different botnet nodes. This is an ongoing rat race that never ends.

2.1.4. Life cycle of a botnet

The life of a botnet starts by infecting a computer to become a bot in the network. Infected bots are connecting to a Command-and-Control server to announce their presence as a bot. This is called calling-home or rallying. With this rallying a communication path is created between the botmaster and the bot. The newly recruited bot can be updated or given commands. If the computer, where the bot software is installed, is connected to the network, the bot software can connect to a Command-and-Control server notify his presence and wait for commands.

The generated network traffic between the bot and the Command-and-Control server is the focus of this research project. The detection of the bot on a network is done by inspecting the traffic coming from the computer on the network. The network packets are evaluated to find the network packets that have the properties that are typical for a bot. The communication between a bot and a Command-and-Control server generates the most traffic in a botnet network. Independent of the type of botnet, most infected computers have the role of being a bot. For that reason, most traffic generated coming from a botnet is between the bot and de Command-and-Control server.

2.2. Machine learning

Learning is something we as a human take for granted. We do it every day. By learning, we get new skills and adapt to changing environments. Doing this in software is a research area that already exists for a long time. A definition of machine learning can be the capability of a computer to learn or adapt to new data. The general forms of machine learning are supervised and non-supervised learning (Zou et al. 2019). With supervised learning, the goal is to train an algorithm to predict the class or value of a data point as close as possible by providing predefined training examples.

With unsupervised learning, the algorithms learn the patterns in the data to separate the data into classes. There are no predefined labels, the algorithm looks at undetected patterns in the data. Supervised machine learning can be used to split the data into multiple classes (classification) or to give a numerical value (regression). Many sub forms of machine learning algorithms are defined that used a combination of supervised and non-supervised learning to get a result or use a combination of classification or regression.

In our case, we use machine learning algorithms to classify a dataset into two classes, one for network packets that are not coming from a botnet and one for network packets that are coming from a botnet. Training an algorithm is done by showing the algorithm data that has been labelled the origin of the data, supervised learning. There can be many methods for labelling data. It can be done manually, or with the use of rules or with the combination of both. There are many methods to label the data and any combination can be used. After an algorithm is trained, new data samples can be shown to the algorithm for evaluation.

The process we use is the following. First data is captured, and the truth is added. Then this data can be used for training an algorithm. After training, newly captured data can be used to be evaluated by the trained algorithm. This process is not adaptive in the way it changes its behaviour during its use. Changing the algorithm can only be done by starting the process from the beginning.
The data that is used for training an algorithm should be carefully chosen. The data should represent all data that will be shown to the trained algorithm during evaluation. If the new data is not within the same range as the data used for training the prediction can be wrong.

The following question is to how to balance the data. Balancing the data means to equalize the ratio between the classes inside the data set. For most circumstances, this is the best options, but not all real live data is balanced.
Balancing the data can be done in multiple ways. Krawczyk (Krawczyk 2016) presents various methods to balance the data, especially for binary classification problems. The problem can be solved on the data level, algorithm level or a hybrid option.

Data level: On data level the two options to balance the dataset are

1. Over-sampling. Increase the size of the smallest class by creating copies of the data.
2. Under-sampling. Decrease the size of the largest class by removing data randomly.

A third option could be a combination of over-sampling and under-sampling.

Algorithm level: The algorithms can be adapted to cope with the unbalanced data sets. Most machine learning algorithms tend to train towards the majority group. To change the algorithm, to cope with unbalanced data, a deep knowledge of the internal method is needed. For most methods, it is difficult to modify the algorithms to handle unbalanced data.

Hybrid option: A third option is the hybrid option. It is a combination of the data and algorithm option. Some methods can be changed to be a little insensitive for unbalanced data but not completely. The data may be unbalanced but not too much.

The bias-variance trade-off is a common problem with machine learning and data sets. (Briscoe et al. 2011). A machine learning algorithm should balance between bias and variance. The bias is the difference between the average prediction and the value that should be reached. With a high bias the model has a high average error. The variance is variability of the prediction or the spread of the data.

With a high bias the algorithm oversimplifies the model. There is more information in the data than the algorithm can model.
A model with high variance is not generalizing the model. The model uses information from the data that is not important. The first case we call "Under fitting" high bias and the second case "Overfitting" high variance. See Figure 2.

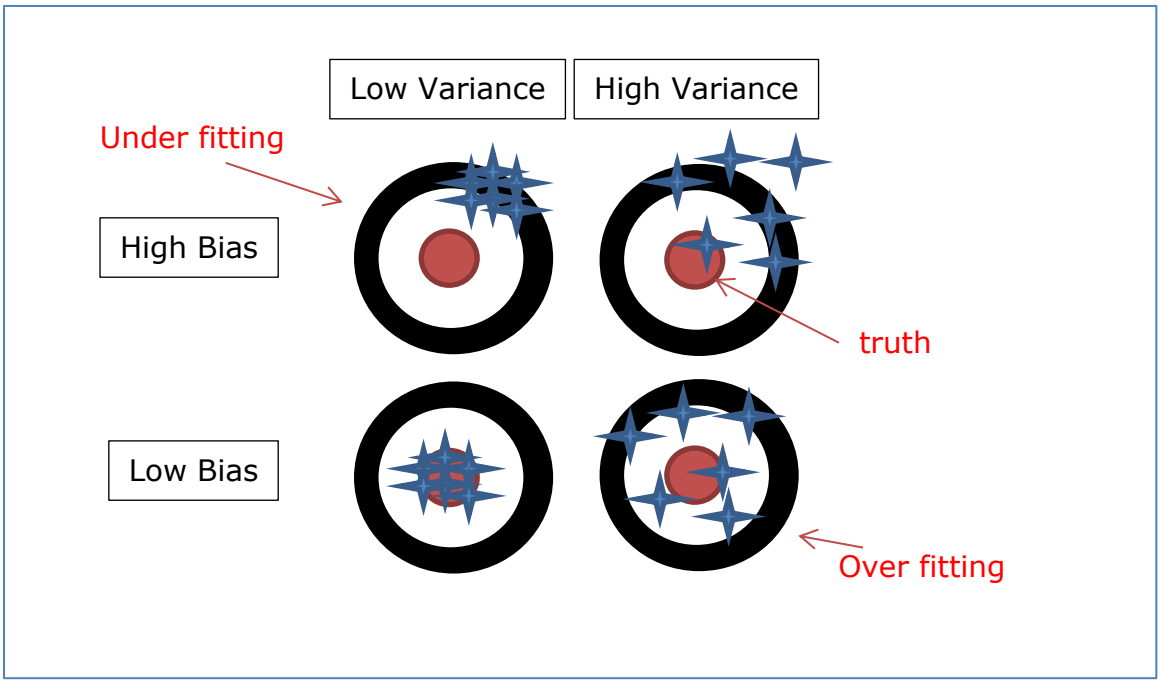


Figure 2: Bias variance trade off (Huilgol 2020)

The bias variance trade-off is the trade-off when optimizing the error of the model against the model complexity. See Figure 3

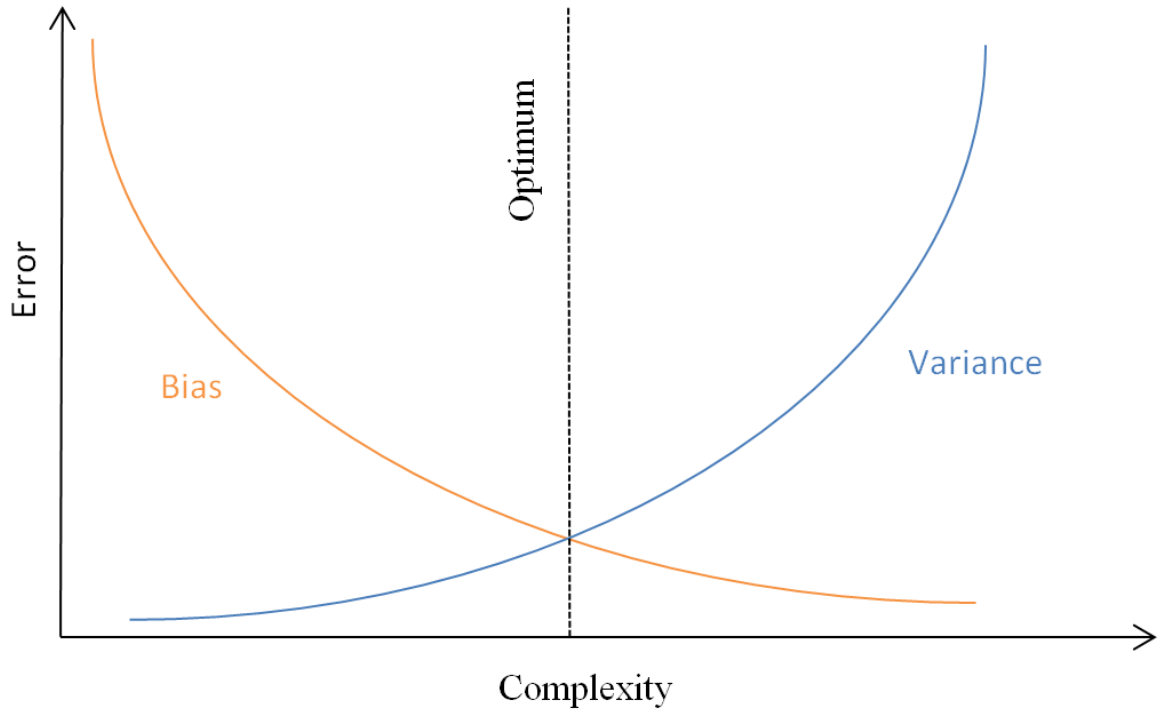


Figure 3: model complexity (Huilgol 2020)

On average, a complex model has a lower bias and higher variance during validation. The model is capable of correctly learning the training data, but during validation, the training was too specific on the trained data. The number of samples was too small for the model, not only the generic features are used.
On the other side is a simple model. This model can only use generic features to model the data, but when the model is too simple, it cannot use enough generic features. The bias error becomes higher. The variance will probably be lower because the model uses only generic features.

To prevent over-fitting of the data, the training data should represent the entire feature space and be large enough. Decreasing or increasing the training set slightly should not give a different outcome of the prediction error. Determining if a model is over-fitted can be done by reducing the model complexity with small decrements and watch if the prediction error increases. When the prediction error is not increasing, directly after the first couple of model complexity decrements, the model is too complex. The same method can also be used to detect under-fitting, when the model complexity is increased, the prediction error should not drop. When the prediction error is not changing all generic features are used and the model complexity is large enough. (Briscoe et al. 2011)

There are numerous machine learning algorithms available, Fernandez-Delgado et al. (Fernandez-Delgado et al. 2014) created an overview of the most popular. From this document, we will discuss the two most promising machine learning algorithms that performs best in most applications. A third machine learning algorithm that gives promising results is from Friedman (Friedman 2001). The three discussed machine learning algorithms are:

1. Support Vector Machine
2. Random Forrest

3. Gradient Boosted Trees

There are many more machine learning algorithms that can be used to evaluate the ability to detect Botnet traffic, but we only discuss these three types.

A trained model for a machine learning algorithm should be validated to know its performance. This process is called model validation. The trained model should be validated with samples that are not used during training. The type of model validation used is cross-validation (Wong 2015). With the cross-validation, we get an error value that represents the predictive effectiveness of the model. There are different types of cross-validation methods, but they have the same process structure.

1. Partition the dataset in a training and validation set
2. Train the model with the training set.
3. Validate the model with validation set.

When a model is created in an iterative fashion, the process above can be repeated. The split of the dataset should be done in such a way that the training and the validation dataset represents all data. (Browne 2000) The splitting of the data can be done in various ways. Some noteworthy examples are, (Wong 2015)

- K-Fold
- Leave one out

With the K-Fold method, the dataset is divided in K number of folds of equal size. One-fold is used for validation and the rest for training. During the training iterations, the validation fold is changed. When for example, there are ten folds and the training has ten iterations, then every fold is nine times used for training and one time for validation. The value of K can be any number between 2 and the number of samples in the dataset. The number of training iterations does not have to be equal to the value of K.

The "Leave one out" cross-validation method, is in essence, a special case of K-fold where K is equal to the number samples in the dataset. This method is mostly adopted when the number of samples in a dataset is low. Because the "Leave one out" cross-validation method is often used, it is named separately.

2.2.1. Support Vector Machine

The Support Vector Machine (SVM) is a machine learning algorithm that can divide object data with features into two classes. The SVM is a binary classifier; it supports only two different classes to be separated. The theory of the SVM is based on the work from Cortes et al. (Cortes and Vapnik 1995). An SVM tries to split the data into two classes using a hyperplane or set of hyperplanes in a multi or infinite-dimensional space. The hyperplane is created by calculating the largest distance between the training data points.

To explain the concept of SVM we start with a simple example. We begin with an example of classifying a dataset in two classes which contain only one feature value per data point. This can be drawn in a one-dimensional space, see Figure 4

Figure 4: One dimensional data features

This is a very simple example, and it is easy to draw a vertical line to separate the two data sets. Suppose there is one blue point close to the orange points, like in Figure 5. It is more difficult to draw a vertical line to separate the classes. The new blue data point is not the same as the other blue data points. This point is an outlier and can cause the model not to work. A new orange data point close to the other orange data points but left from the right blue point will be misclassified, although it belongs to the orange class. The resulting threshold between the classes creates an algorithm that has a high bias. Placing a threshold, without taking outlier into account, is called a hard margin.

Figure 5: Outlier

Although there is a blue point close to the orange points, the threshold between the classes is the same as in Figure 4. The new blue data point does not change the separation value. Allowing the SVM to make mistakes and have a margin as large as possible is called a soft margin. The soft margin helps to generalize better on unseen data increasing the performance of the SVM.

In a SVM algorithm, the threshold value is created such that the distance from the different data points from the different classes to the threshold value is maximal. When a hard margin is used, the distance from the closest points in both classes is determining the location. The distance from this point to the threshold is called the support vector. In our case, we have a one-dimensional system, but this can be easily extended to multiple dimensions. Because a hard margin is not always the best method, support vectors can be calculated using soft margins. The soft margin is not one fixed threshold but a threshold with a plus and min margin. This is an optimization problem that is using cross-validation to find the best optimum, minimizing the error.

Using the one-dimensional problem in our example is easy to solve. But when we extend our data with new samples, the problem can be more difficult, see Figure 6.

Figure 6: Extended data

It is clearly visible that there are three groups of data and that one threshold is not solving the problem. This can be extended to an almost infinite number of data groups and classes. An SVM solves this by extending the dimensions of the data to separate the classes. In the above case, we can create an extra dimension by taking the square root of the data point for the second dimension, see Figure 7.

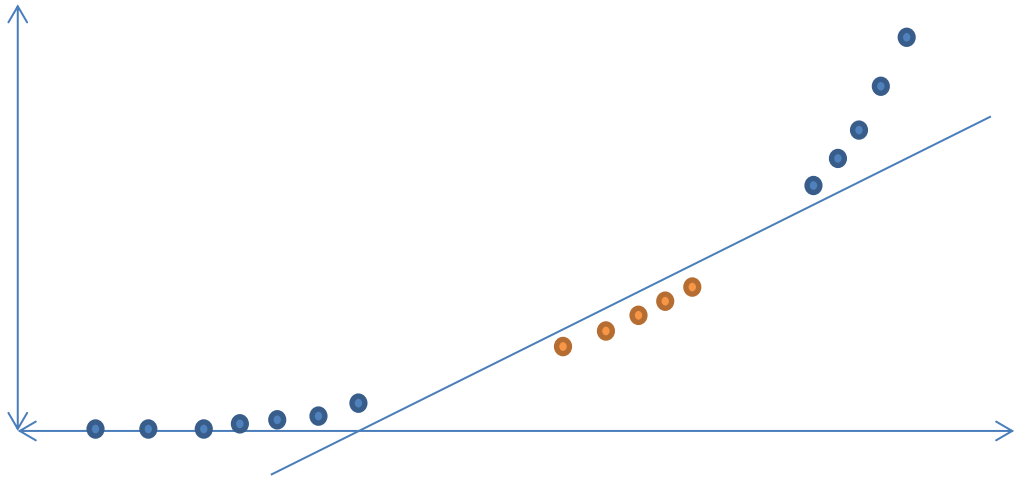


Figure 7: Extra dimension

The new threshold is now not a value but a line. With the line, it is possible again to separate the two classes and evaluate the class of new measurements. The number of dimensions the data can be extended is unlimited. When needed for separation, the SVM can add a third or a fourth dimension.

In this example, we use the square root of the data to calculate a new dimension. It can be imagined that there are many functions possible to calculate a new dimension from the data. In the original work from Cortes et al. (Cortes and Vapnik 1995) the extra dimensions are created from linear functions resulting in a linear classifier. Using the kernel trick (Aizerman et al. 1964), it is possible to create a nonlinear classifier efficiently. The calculations are the same; only the data vector is transformed. With the kernel trick, an infinite number of dimensions can be used but the calculations needed are not more than the number of dimensions in the data vector, reducing the number of calculations drastically. Many Kernel functions are proposed. Some often-used examples are:

- Gaussian Kernel
- Gaussian Kernel Radial Basis Function
- Sigmoid Kernel
- Polynomial Kernel

Selecting the right kernel can be difficult and time-consuming (Ali et al. 2006). The general approach is to try the simple one first, linear, and test more complex kernels afterwards. If the results do not improve, use the simplest kernel for new data evaluation. More complex kernels often need parameters to be optimized for the data that is used. Optimizing this parameter is often an iterative process where the cross-validation is used to select the best value for the parameter.

2.2.2. Random Forest

The random forest machine learning algorithm is based on a forest of decision trees (Ho 1995), (Breiman 2001). To explain how a random forest algorithm works, we first need to know how a decision tree works. First, build a tree before you can think of making a forest.

A decision tree is a binary tree that exists of a root, the first node where everything starts, internal nodes and leave nodes, see Figure 8.

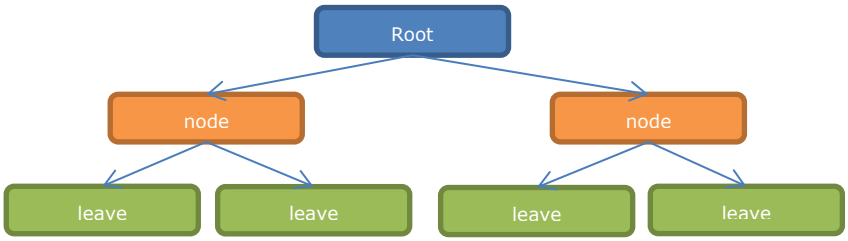


Figure 8: Decision tree

With the root and internal nodes, a question is asked that can be true or false. This can be a question that directly has an answer true or false or the question can be an expression like is a value smaller than 10, v < 10. Asking the right questions will give the answers that lead to a leaf that gives the correct class or value.

In a dataset, every data point has multiple variables. A data point has the following form

$$(x, Y) = (X_1, X_2\ X_3, \ldots . X_k, Y) \qquad \text{Eq 1}$$

x represents the different values of a data point and Y the target value. For every x, a question is asked in the decision tree. What question should be asked, and which one is first is the next step. The first question is that question that separates the dataset best in the different classes. There are some techniques to determine the best question. Commonly used methods are Gini index and entropy

Gini index $$\mathrm{G(E)} = 1 - \sum_{j=1}^{C} \mathrm{p}_j{}^2 \qquad \text{Eq 2}$$

Entropy $$\mathrm{H(E)} = -\sum_{j=1}^{C} \mathrm{p}_j \log \mathrm{p}_j \qquad \text{Eq 3}$$

Where 'C' is de number of classes and 'P' the probability a data point is classified correctly for this class.

Some advantages of decision trees are that they are simple to understand and implemented. Also, they can be used with large non-linear datasets. The downside of decision trees is that a small change in the training data can result in a completely different tree, producing a different outcome. In general decision trees are good in predicting the trained data, but when new data, that is different, needs to be classified a decision tree is not always accurate.

Now we know what a decision tree is we can extend this to a random forest. At first, a bootstrapped dataset is created from the original dataset. The new dataset has the same size as the original one only the samples are taken at random from the complete original dataset. This means that not all samples from the original set are used, and some samples are used multiple times. This is called boosting. From this new dataset, a decision tree is created. But instead of selecting the best feature for the first question in the decision tree, the feature is selected at random. This process can be repeated multiple times, creating multiple decision trees. In a normal application, the number of trees can be large, hundreds or thousands. When a new data value needs to be evaluated, the

data value is evaluated with all the different decision trees. For a classification random forest, the outcome will be the majority vote of all trees. If a value is requested, the average of all trees is used.

To classify the quality of the random forest classifier, an Out-of-Back error can be calculated. When a decision tree is created, not all data is used from the dataset. This data that is not used is called Out-of-Back. The Out-of-Back data can be evaluated with the tree where it is not used. The Out-of-Back error is the percentage of samples that is misclassified from all Out-of-Back samples. The creation of a random forest can be repeated with the same data. Eventually, the random forest with the smallest Out-of-Back error is the random forest that will be used.

A random forest classifier can handle large dataset. Values in the datasets can be used as is, there is no need to scale them. Also, a random forest will not overfit to the data.

2.2.3. Gradient Boosted Trees

Gradient boosted trees are based on the hypotheses that the combination of many weak classifiers will result in a strong classifier, (Friedman 2001). A weak classifier is a classifier that is performing poorly and is unable to create a meaning full prediction when new data points are evaluated. A weak classifier is only performing a little bit better than a random answer. A gradient boosted tree classifier consists of many small decision trees, see Random Forest classifier, section 2.2.2. The used decision trees have a predefined maximum number of leaf nodes. Depending on the number of features used in the dataset, the number is between 4 and 32 leaf nodes. The number of nodes is almost always a number that is a power of 2. (2, 4, 8, 16, 32 etc.). Because we are creating a weak classifier, a larger number than 32 leaf nodes are not often used.

When a decision tree is created, and a new data value is evaluated, the outcome of all trees are added together to get a prediction value. The tree is added with a fixed, lower than 1, gain value, the learning rate. Every tree is contributing only partly to the answer, see Figure 9. A learning rate is used to lower the variance of the gradient boosted tree.

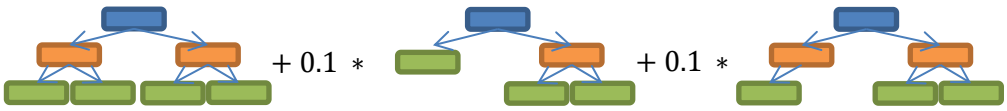


Figure 9: Gradient Boosted Tree

A gradient boosted tree is created by first boosting the dataset as explained in Random Forest, section 2.2.2. The boosted dataset is used to create the first decision tree. The first tree consists of only one leaf and is the average value of all target values. When the first tree/leaf is created, all data values in the training set are evaluated. For every data value, an error is calculated called residual.

$$\text{residual} = V_{target} - V_{predicted} \qquad \text{Eq 4}$$

where:

$V_{target} = target\ value$
$V_{predicted} = predicted\ value$

This residual is used in the next decision tree as a target value. A new boosted dataset is created, and with the residual, a new decision tree is calculated. This procedure continues till the total residual is not changing any more, is zero or a previously defined maximum number of trees is used.

The above description is valid when Gradient Boosted Trees are used to predict a regression value. For a binary decision, some small modifications are needed. A false and true value be a 0 and 1. The first initial prediction will be the log of the odds.

$$initial\ prediction = \log\left(\frac{Number\ of\ true\ data\ values}{Number\ of\ false\ data\ values}\right) = \log(odds) \qquad \text{Eq 5}$$

To calculate the residual of the prediction, the probability is used.

$$probability = \frac{e^{\log(odds)}}{1-e^{\log(odds)}} \qquad \text{Eq 6}$$

With the definition of the residual, a new decision tree can be created. In a gradient boosted tree, the leaves are a list of residuals. For classifying, gradient boosted trees the residual cannot be used directly. A prediction cannot be added directly to a probability; there is a transformation needed. A generally used transformation is:

$$prediction = \frac{\sum residual_i}{\sum[prev\ Probability_i * (1-prev\ Probability_i)]} \qquad \text{Eq 7}$$

where:

$$\sum residual_i = sum\ of\ all\ residuals\ for\ a\ leaf_i$$
$$prev\ Probability_i = previous\ Propability\ for\ a\ leaf_i$$

The prediction of a leaf is the sum of the residuals of all data points in the training set where this leaf is an endpoint divided by the sum of a previous probability of the data points times one minus this previous probability.
With these formulas a gradient boosted tree for regression can be used for classification.

A big advantage of a gradient boosted classifier is the accuracy. There is no need to pre-process the data and even performs accurate when the training data is not completely representing the total domain. Overfitting is a problem with gradient boosted classifiers. When the training set is too small, the training samples are learned. Another issue with gradient boosted classifiers is that they are resource intensive. The number of decision trees can be very large, over 1000, consuming a lot of memory and processor cycles.

2.3. Effectiveness

The effectiveness of a classifier can be calculated using different methods. The first method is the accuracy (Sokolova et al. 2009). The accuracy is most used to quantify the effectiveness of a classifier.

$$Acc = \frac{TP+TN}{TP+TN+FP+FN} \qquad \text{Eq 8}$$

The accuracy is based on the values in a confusion matrix, see Table 1.

		Actual	
		True	False
Prediction	True	True Positive (TP)	False Positive (FP)
	False	False Negative (FN)	True Negative (TN)

Table 1: Confusion matrix

The accuracy can be easily used to compare classifiers, but only when the distribution of the two classes is the same in the used datasets. When the distribution of the classes inside the dataset changes, the accuracy also changes. Therefore, an accuracy number can only say something about a classifier when the same dataset is used when comparing classifiers.

A better method of comparing classifiers can be the use of the Matthews Correlation Coefficient, (MCC) (Matthews 1975). This coefficient is better for comparing different classifiers when the distribution of the input data is different.

$$MCC = \frac{TP*TN-FP*FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \quad \text{Eq 9}$$

Because the MCC is considering the balance of the two classes inside the dataset, the MCC is a better value to compare classifiers. The MCC is not often used because comparing classifiers trained on different dataset is not meaning full. The MCC is given for completeness.

A third method of giving the effectiveness of a classifier is the ROC-curve, (Fawcett 2006). De ROC is a graphic where on the y-axis, the True positive rate (TPR) is shown, and on x-as the False positive rate (FPR).

$$TPR = \frac{TP}{TP+FN} \quad \text{Eq 10}$$

$$FPR = \frac{FP}{FP+TN} \quad \text{Eq 11}$$

The outcome of a classifier is often a value between 0 and 1. A threshold determines if the value is a False or True. When changing the threshold value from 0 to 1 in small increments, the ROC curve can be drawn, see Figure 10.

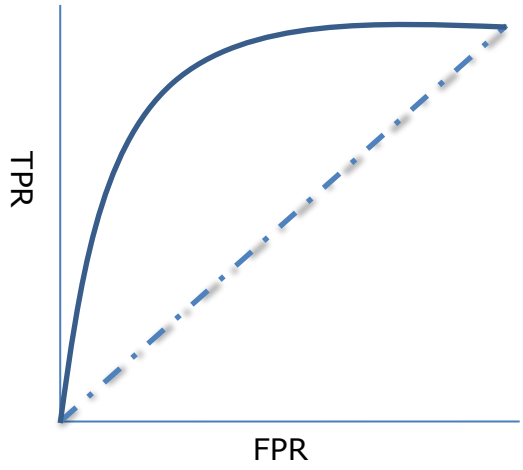


Figure 10: ROC Curve

A classifier that produces a straight line (dashed line) is performing the same as a random generator. The closer the line is to the left, top corner (solid line) the better the classifier performs.

2.4. Network Flow

In this document we have often talked about network flows or only flows. What are network flows? In our definition flows are a group of network packets that belong to each other. This definition is still large and can be split in two depending on the type of network packets. The two different network packets are TCP/IP and UDP/IP (rfc 1981).

For TCP/IP a flow is defined as one TCP session. A TCP session is started by a client that asks a server to open a communication channel for transferring and receiving data. When the client or server is done transferring data, it can close the communication channel which closes the TCP session. A network flow is all traffic from a TCP session. The TCP protocol is using acknowledges to make sure the data transfer between client and server is guaranteed. When data is lost during transmission the protocol assures a retransmit of the data. For our research, a network flow should discard the lost messages. If not discarded, network flows will change when the network topology is error prone.

When using TCP/IP the network flow has a good definition. For UDP/IP this is different. There is no communication channel opened. There is no fixed start or stop sequence. To be able to create a UPD/IP network flow, the following definition is used. The first network package between a client and a server that uses a send and receive port number that is new will be used as the start package for the network flow. All network packets that used the same port numbers will be included in this network flow. A flow will be marked as ended if there are no network packets received within a pre-defined timeout value. For example, if there are no network packets received within X minutes of the last network packets that uses the same send and receive port number the flow is ended. When a new package arrives after X minutes with the same send and receive port number this will start a new flow. For some protocols on top of UDP the sequence of packets is known. These protocols use fixed port numbers that can be used to track the beginning or ending of a flow. Because the list of fixed port numbers is large it can be an enormous task to implement all protocols on top of UDP. But for some protocols that are used extensively in a network environment this can be worth the effort.

Because for UPD/IP there is no defined end or start, it can be difficult to create the flows. The flows are only ended after a certain timeout value. This can make the list of flows that are not closed very large. Software tracking network packets to create flows should take care to do this efficiently.

3. Method

The answer to the research question can be found by first answering the sub-questions. Every sub-question has a different method for getting the answer. The technique for every sub-question is given in the next sub-chapters.

RQ1: Which dataset will be used to train and evaluate machine learning algorithms?

Most machine-learning algorithms need a large dataset for training and validation. The dataset should be representative of the real-world problem and accurate enough, without many classification errors (Cortes, Jackel, et al. 1995). Generating such dataset can be a huge task and a research project of its own. There are already datasets available that can be used for this research project. A dataset should contain enough data to calculate the needed features for training. When selecting multiple datasets, it should be possible to create the same feature-set from all datasets.

RQ2: Which training features, from TCP/IP network traffic, should be used for botnet detection?

Creating features from the dataset is a non-trivial task. The work from van Klaveren (van Klaveren 2019) is used to find the first set of features. This can be used as a baseline performance. When there is more knowledge about the used features and how important they are for the algorithm, we can evaluate new features. Good understanding of the TCP/IP protocol and the working of a botnet are necessary to find or calculate new features from a dataset. The training phase will help to evaluate the importance of a specific feature. Selecting the best features from deep learning neural networks can be complicated, as shown by van Klaveren (van Klaveren 2019). For more classical machine learning technologies like SVM and Random Forest, it is easier, (Chandrashekar et al. 2014), (Ho 1995). Features extracted from a dataset can have high importance for an algorithm to get the desired classification. But these features can also be biased to the dataset. An example of this case can be the TCP/IP destination address of a network package. For certain dataset, this can help the trainer to get better performance, but by the nature of a botnet, the value of this feature will change during time, making the algorithm unusable. Each feature selected should be independent of the changes in a network topology. Features extracted from network time and destination should be handled carefully. For every feature that is used to build a training set, we should ask ourselves if the feature is independent of the network topology. If features are changed easily by a changing environment, the feature should not be used. When the best performing feature-set is known, the feature-set can be reduced, by removing the features that do not contribute much to the result of the algorithm.

The datasets used to create features can also have a class imbalance. This means that there are more examples of one class compared to the other class. This imbalance in the dataset can have an impact on how the model is trained. When more examples of one class are present, the model will likely select that class more than the other class. Two methods to undo the imbalance is to over-sample the smallest class or under-sample the largest class (Batista et al. 2004). Both methods will be used to train a model. Because the used datasets are very large, we do not expect large differences between the methods.

RQ3: What selection of machine learning algorithms can be used best to detect botnets?

There are many machine learning algorithms that can be used to classify the dataset. Fernandez-Delgado et al. (Fernandez-Delgado et al. 2014) already give a selection of 179 classical classifiers, and then there are a large number of neural network technologies that can be used. We will choose two types of classical classifiers that performed best according to Fernandez-Delgado et al. (Fernandez-Delgado et al. 2014), Support Vector Machines (SVM) and Random Forest Classifier (RFC), and one from Friedman (Friedman 2001), the Gradient Boosted Trees (GBT). The idea is that with a reduced feature-set as used by van Klaveren (van Klaveren 2019) a classical classifier can also be successful. Most recent research use a neural network to do the classification. Another reason to use a more classical classification algorithm is the resources it uses. Neural networks are mostly very resource-intensive (Widanapathirana et al. 2012), which can be a problem in some situations. We will use Python (Python 2020) as a programming language because there are many libraries available that have implemented different classifiers. Scikit-learn (scikit-learn 2020) is an example of a library that has many classifiers available. Scikit-learn support SVN, RFC and GBT, making it easy to evaluate the different algorithms. What can be a problem is the size of the dataset that is used. Because Python is an interpreted language, it can be relatively slow. For handling large amount of data Java (Oracle 2020) is a better language to use (Aruoba et al. 2014). In our case we use Java to do the conversion of the data from raw network packets to features. And we use python to create machine learning algorithms. In this way we can use the speed of Java and the flexibility of python where needed.

RQ4: How can the different algorithms be compared in their effectiveness and efficiency in detecting botnets?

Many performance parameters can be calculated to express the effectiveness of an algorithm. But not all metrics are useful when different datasets are used. When two algorithms use the same dataset the accuracy, (Sokolova et al. 2009), is the most used method to express the effectiveness of a machine learning algorithm, but there exist more methods. Besides the accuracy, the Matthews Correlation Coefficient (MCC), (Matthews 1975), and ROC graph, (Fawcett 2006), will be calculated to express the effectiveness of the algorithms. The trained algorithm is also evaluated with a dataset that is not used during training. This gives an idea of how the algorithm performs on new data simulating a more realistic environment.

When the effectiveness of the algorithm is known, an efficiency parameter should be given. There are two cases where the efficiency can be given.

1. Training
2. Validation

There are different ways to express the efficiency of an algorithm. It can be time or used energy, the cost in money or the needed computer cycles. Some are easy to calculate, like time and others are more difficult, for example, the cost. It is important that the calculation of efficiency is simple to obtain and can be reproduced without external hardware. The efficiency can be challenging to

express. Differences in the layout of a computer can be significant for the efficiency of the computer. Does the computer have specialized hardware to speed up the computations? For example, a graphics card to do the training and evaluation. Or a large amount of memory so the complete dataset can be loaded into memory completely, without the need to get it from a slow storage device. For the training phase of the algorithm there are many variables that are influencing the efficiency of the algorithm. It will be difficult to give a number that can be used in other papers for comparison. Therefor we give the relative time the algorithm uses to perform a training cycle. The slowest algorithm gets the value 100. With the following formula Eq 12 the value of the faster algorithms can be calculated.

$$eff_t = \frac{time_t}{time_s} \qquad \text{Eq 12}$$

where:

$eff_t = training\ efficiency\ of\ the\ algorithm$
$time_t =$ time to train the algorithm
$time_s =$ time for the slowest algorithm to train

The evaluation efficiency of an algorithm can be expressed as the time needed to evaluate one action. This means that the output of one feature-set is calculated and the time needed is given. The given time should be compensated with the speed of the computer where the algorithm is evaluated. With this number, it should be possible to get a number that can be used in other papers for comparison. Formula 2 can be used to calculate the efficiency.

$$eff_e = \frac{1}{time_e * spec} \qquad \text{Eq 13}$$

where:

$eff_e =$ the evaluation efficiency of the algorithm
$time_e =$ the time needed to perform the one action
$spec =$ speed of the computer in GFlops

The GFlops number can be calculated with the LINPAC benchmark (Dongarra 1988). LINPAC is available for many computer architectures and often used as a measure for performance. LINPAC is not without controversy, because it gives the maximum performance of a computer and not the usable performance. There are many factors influencing the performance of a computer that LINPAC discards. The upside is that LINPAC is giving one number that can be used in our calculation. To be sure the research can be repeated, and the outcome used in other papers, the layout of the used computer should also be given. The factors that could have influenced the calculation can be pointed out later.

A third important efficiency parameter is the usage of memory during evaluation. Because we are using python as a programming language and Scikit-learn library for the training algorithm, we do not how optimized the software will be. For example, the SVN algorithm is implemented using a C++ library, which can be very efficient. Therefore, we do not report the memory usage of the used algorithms.

3.1. Validation

In this section, the deliverables and validation for all the questions are discussed and evaluated. The answer to the research question is given by the answers on the four sub-questions. Every question has deliverables that can be used to answer the next question. For the first question, the deliverable will be a dataset that consists of three parts. One part is network traffic that is generated by botnets. The second part is network traffic that is from regular applications. And a third part is network traffic from botnets that are not used during training. This to evaluate the proposed algorithm and simulate a more realistic environment. The created datasets can also be used by other research to validate our results or test new algorithms.

If the usage of the data and method how to use the data, is documented, it should be easy to get the same results as a starting point. It should be possible to produce the same feature-sets from all used datasets. If this is not possible, a dataset is not usable. The datasets used are specially created for the detection of botnets. Recent researchers are using the same datasets to validate their algorithms. An example is from Alauthaman et al. (Alauthaman et al. 2018). This research is also using an unseen dataset to do the evaluation.

From the datasets, features will be selected to be used for training. It should be clear how to calculate every single feature. Every feature should be independent of the network topology and timing. Imbalance in the dataset can influence the training. By applying a balancing technique, as described in heading 2.2, a balanced dataset, is used for training a model. Krawczyk (Krawczyk 2016) classified three areas where the data imbalance can be solved. Data-level, Algorithm-level, and Hybrid methods. Only the Data-level method is applicable.

There will be three different machine learning techniques evaluated. By using a standard library, the chance of mistakes is minimized. The selected algorithms (SVN, RFC, GBT) are one of the best non-neural network machine-learning algorithms. There could be others, but the availability of a library where these algorithms are implemented makes it easier to test the algorithms quickly.

The effectiveness of the used algorithms is compared using different parameters. The accuracy is often used to express the effectiveness of classifying algorithm. The problem with the accuracy can be the dataset imbalance. Because we apply the algorithm to a balanced dataset, the accuracy can be compared more easily. Other parameters, like the MMC, are less influenced by a dataset balance. The MMC is also calculated, although it is not often used in other papers. Because we use a binary classifier there a lot parameters that can be calculated, (Sokolova et al. 2009). A ROC graph will also be calculated to express the performance. All these parameters should help show the effectiveness of the algorithms. (Fawcett 2006).

The efficiency of an algorithm is more difficult to express. For the training phase of the algorithm, only a relative number will be given. There are too many variables that influence an absolute computer performance number. For the validation phase, most variables can be ruled out. Therefore, an absolute number can be given. There are still remarks about these results. First, the implementation of the algorithms can be optimized differently. There is no guarantee that the most efficient method is used to code the algorithm. Secondly, a LINPAC number is used to express the speed of the computer. This number specifies the performance of the used CPU and memory. Specialized hardware, that can speed-up the calculations, is not taken into account (Dongarra et al. 2003).

4. Datasets

Training a machine learning algorithm requires datasets that contain truth labels. Every entry in the dataset should have a label telling which class it belongs. Creating enough data with truth labels can be a huge task and a research project of its own. Models for machine learning algorithms to detect botnets from network traffic can only be created when datasets containing network packets or flows are labelled to which class they belong. Collecting and labelling these datasets can be a lot of work. The number of network packets should be large enough be representative. From other research groups, there are multiple datasets available that have labelled truth values.

The selected datasets can be used to test different algorithm and evaluate how they perform. For this research, five different datasets are used to assess the performance of the algorithms. In Table 2 the datasets are summarised. The first column with number of network packets gives an indication how many packets need to be converted to flows. For training and validate an algorithm only network flows are used.

Dataset	No. network packets	No. of TCP botnet flows	No. of TCP program flows
PeerRush dataset	1,308,280,472	1,901,054	3,538,424
ISOT	161,824,341	8,887	838,236
ISCX IDS 2012	119,961,845	29,840	2,602,302
ISCX Bot 2014	14,502,784	81,580	207,172
CTU 13	74,661,384	61,918	0
Total	1,679,230,826	2,083,279	7,186,134

Table 2: Used Dataset

4.1. Peerrush

The Peerrush dataset (Rahbarinia et al. 2014) is divided into three groups, called D1, D2 and D3. The first group is ordinary P2P traffic. The second group contains the botnet P2P traffic, and the last group is non-P2P traffic from ordinary applications. The data is captured between October 2007 and November 2011.

D1 - ordinary P2P traffic

The traffic from 5 different popular P2P applications is captured. Most captured data is coming from Skype, where the traffic is collected over a period of 83 days. The other applications are eMule, µTorrect, Frostwire and Vuza. Together these programs create a diverse collection of normal P2P network traffic.

D2 - botnet P2P traffic

The P2P botnet traffic is obtained from other sources and contains captured data from three different botnets: Storm, Waledec and Zeus. These botnets are relatively old and not active anymore. The Storm botnet was active around 2008 and spread itself by email. In 2008 the Storm botnet was responsible for about 8% of the total spam created on the internet. The creators of the Storm botnet are never identified. The botnet called Waledec infected around 70,000 to 90,000 computers before it was taken down by Microsoft in 2010. Zeus had his peak

activity in 2009. The creator of the software was arrested in 2013 after research from the FBI. Because we focus on TCP/IP network traffic, only the Waledec is used. All other botnets are using UDP/IP.

D3 - non-P2P traffic
To be able to distinguish regular network traffic from P2P botnet traffic, a network capture tool is used to capture the daily traffic from a network. Daily traffic also consists of P2P traffic. Because the researchers want to classify P2P traffic from regular traffic, all captured P2P traffic is removed from this set. No botnet data is inside this dataset.

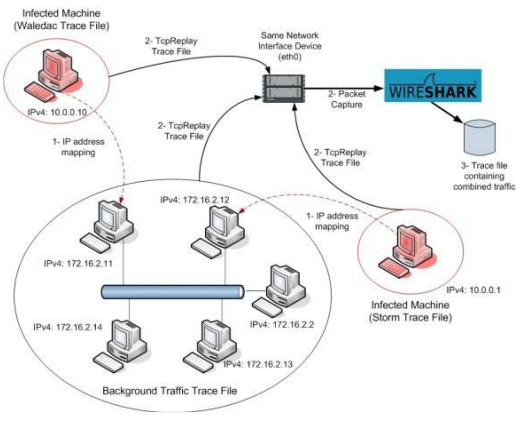
Truth Labels
The total collection of the Peerrush dataset contains multiple pcap files. The pcap files are unlabelled, and no field is telling if a network packet is from a botnet. To be able to label the data, a list of IP numbers is supplied that specify the data coming from botnets. All other data is from regular applications.

Contents

Program	Botnet	IP Type
Skype	No	UDP/TCP
eMule	No	UDP/TCP
uTorrent	No	UDP/TCP
Frostwire	No	UDP/TCP
Vuze	No	UDP/TCP
Storm	Yes	UDP
Zeus	Yes	UDP
Waledac	Yes	TCP

Table 3: Contents of Peerrush dataset

4.2. ISOT



The ISOT dataset (Saad et al. 2011) is a combination of publicly available datasets with malicious and non-malicious data. The dataset originates from different research projects. From the French chapter of the honeynet project, from the Traffic Lab at Ericsson Research in Hungary and the Lawrence Berkeley National Lab. The total dataset contains network traffic form the Zeus and Waledec Botnets and regular network traffic. The regular traffic includes HTTP web browsing, World of Warcraft gaming packets, and packets from popular BitTorrent clients such as Azureus. The different public datasets are replayed in an isolated, controlled environment and captured again using Wireshark (Wireshark 2020). The dataset is replayed to have a different network environment than the original dataset but can be easily classified being Botnet traffic or not. Having a dataset coming from a different network environment is important to validate if an algorithm to detect botnet traffic is independent from

the network environment. The data is captured between October 2004 and October 2007.

Truth Labels
The ISOT dataset contains around 3.3% malicious traffic and 96.7% non-malicious traffic flows. Traffic flows are a collection of network packets that belong together. The labelling of the data is done by specifying specific IP and MAC addresses as malicious. This knowledge will be used to create training and validation datasets.

Contents

Program	Botnet	IP Type
VOIP	No	UDP
HTTP WEB	No	TCP
Bittorrent	No	TCP
Background traffic	No	UDP/TCP
Storm	Yes	UDP
Waledac	Yes	TCP

Table 4: Contents of ISOT dataset

4.3. ISCX IDS 2012

The ISCX IDS 2012 dataset (Biglar Beigi et al. 2014) originates from the University of New Brunswick (UNB). Most, publicly available, datasets have issues. They are sub-optimal because not all internal data may be shared due to privacy issues. For example, the payload of the network packets can contain personal information. Using suboptimal datasets can give problems in the validity of the research. When network packets are removed from the dataset because of privacy issues the trained algorithm can have problems classifying them when needed in a real-world environment. The ISCX IDS 2012 dataset contains dynamically created data that better reflects real-world traffic. The dataset contains traffic from HTTP, SMTP, SSH, IMAP, POP3, and FTP. In total, the data of 7 days of activity is captured.

- Friday, 11/6/2010, Normal Activity. No malicious activity
- Saturday, 12/6/2010, Normal Activity. No malicious activity
- Sunday, 13/6/2010, Infiltrating the network from inside + Normal activity
- Monday, 14/6/2010, HTTP Denial of Service + Normal Activity
- Tuesday, 15/6/2010, Distributed Denial of Service using an IRC Botnet
- Wednesday, 16/6/2010, Normal Activity. No malicious activity
- Thursday, 17/6/2010, Brute Force SSH + Normal activity

The IRC Botnet activity is coming from an Internet Relay Chat Botnet specially created to be added in this dataset.

Truth Labels
The ISCX IDS 2012 dataset includes truth files in XML format. The truth is given using flows. When the captured packets are converted to flows, the truth can be

evaluated. By comparing the protocol type, source, destination addresses, and port numbers the truth label can be found in the XML file.

Contents

Program	Botnet	IP Type
HTTP	No	TCP
SMPT	No	UDP/TCP
SSH	No	TCP
IMAP	No	TCP
POP3	No	TCP
FTP	No	TCP
IRC botnet	Yes	TCP

Table 5: Contents of ISCX IDS 2012 dataset

4.4. CTU 13

The CTU-13 (García et al. 2014) is a dataset of botnet traffic that is captured at the CTU University, Czech Republic, in 2011. All data that is made public contains only the botnet network packets. All regular traffic is removed from the files due to privacy reasons. For training a botnet detector, this dataset should be combined with datasets that include regular network traffic. The total dataset contains 13 scenarios where botnet traffic is captured. The scenes include traffic from 7 different botnets, caught in a pcap file format. In total, this dataset contains more than 70Gb of data. The botnets in this dataset are Neris, RBot, Virut, Menti, Soguo, Murlo and NSIS.ay.

Truth Labels

The truth values of this dataset are easy. All data that is published is coming from botnets. Because we do not differentiate between the different botnets, all data can be labelled the same.

Contents

Program	Botnet	IP Type
Neris	Yes	TCP
RBot	Yes	TCP
Virut	Yes	TCP
Menti	Yes	TCP
Soguo	Yes	TCP
Murlo	Yes	TCP
NSIS.ay	Yes	TCP

Table 6: Contents of CTU13 dataset

4.5. ISCX Bot 2014

This dataset (Biglar Beigi et al. 2014) is created using three other datasets. The aim is to create a general, real, representative dataset. The dataset includes ISOT, ISCX 2012 IDS dataset and Botnet traffic generated by the Malware Capture Facility Project (CTU 13). There are two capture files, one for training and one for testing. The complete set contains traffic from various botnets:

Neris, Rbot, Menti, Sogou, Murlo, Virut, NSIS, Zeus, SMTP Spam, UDP Storm, Tbot, Zero Access, Weasel, Smoke Bot, Zeus Control and ISCX IRC bot. Around half of the network, packets are from normal flows, and half are from malicious flows.

Truth Labels
The truth values are created by specifying the source and/or destination address of the flow. When a flow is created, the IP addresses can be used to label the data. Flows with IP addresses that are not specified are from standard applications.

Contents

Program	Botnet	IP Type
ISOT		
ISCX 2012 IDS		
Neris	Yes	TCP
RBot	Yes	TCP
Menti	Yes	TCP
Sogou	Yes	TCP
Murlo	Yes	TCP
Virus	Yes	TCP
NSIS	Yes	TCP
Zeus	Yes	UDP
UDP Storm	Yes	UDP
TBot	Yes	TCP
Zero Access	Yes	TCP
Weasel	Yes	TCP
Smoke bot	Yes	TCP
Zeus Control	Yes	TCP
IRC Bot	Yes	TCP

Table 7: Contents of ISCX Bot 2014 dataset

4.6. Remarks

The total amount of data of all combined datasets is more than 360 GB of disk space. This data comes from different sources but is closely related. The datasets overlap partly and include the same data. Some datasets include the same information but try to differentiate by replaying the network traffic in a slightly different environment. In Appendix A, a diagram is constructed to visualize which botnets are included in which datasets. For this research project, all data is used, and it is accepted that there is double data. Not only the botnet network traffic is double also data from regular traffic can be double. The use of all data can influence the outcome of the research, but I did not investigate this issue. Another issue in this data can be the imbalance of the different botnet network data. Some botnets are included with a more significant amount of traffic data than others. Because we do not differentiate between the different botnets and only have a binary truth label, we do not consider the imbalance in the traffic from the different botnets. Because we do not investigate this imbalance, we do not know the impact on the results.

As can be seen from Table 2 the datasets are unbalanced. The number of flows coming from normal network traffic is larger than the number the number of flows coming from botnet network traffic. This unbalance should be solved during training of the algorithms.

The last remark about the datasets is the age of the different sets. Most datasets are more than ten years old. Most botnets that are inside the datasets are not active anymore. Even when they are not taken down actively, we should assume they have evolved and are entirely different right now. Ten years is a long time for software.

5. Botshot

The datasets are all using the pcap file format to store captured network traffic. This file format does not include any information about traffic type, botnet or regular, or the flow of the traffic. With flow, we mean the network packets that belong to each other and together form a communication session. The training and validation sets can be created from the datasets with the selected flow features. When flow features are extracted, a machine learning algorithm can be trained and validated. This process is graphically displayed in Figure 11.

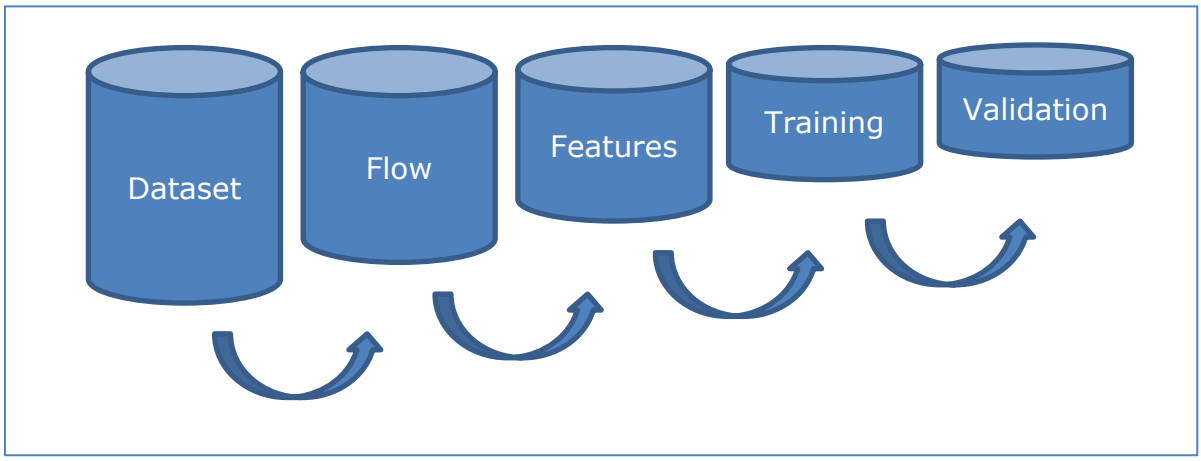


Figure 11: Data processing flow

Every step is reducing the amount of data considerably, making processing easier. From Datasets to Flows the payload of the packages is dropped. The payload will not be used because it is easy to encrypt and then it is useless to inspect. From Flow to Features, all double information can be skipped. An example is de source and destination address. It is present in every packet in the flow but has the same value. Not all features calculated are needed for training. So, the training set will consume less space than the Feature set. Also, the Feature set is split in a training and validation set. Where the training set is most likely be larger than the validation set.

Almost all research for detecting botnets in network traffic flows has implemented a similar kind of data processing method. Doing this manually is very complicated and time-consuming, so doing this automatically with software is preferable. Many researchers have created software to implement this kind of data processing method. Some have used commercial software to create the flows. Most of this software is part of a complete security package, expensive and difficult to use for our problem. The reuse of software would be preferable because it saves time, but I did not find software that was usable for this research to implement the data processing flow. Software could create flows but was not able to implement the truth label into the flow. Some software was only available for one dataset. And when sources where available there was no documentation about the software (van Roosmalen 2017; Poon 2018). Adapting existing software to be able to process all datasets and be reusable was difficult, therefore, a new software package is designed to perform the data processing.

To make sure the software is reusable by others the software has been made extendable and capable to be changed easily when new unknown, dataset becomes available.

The main properties of the software are.

1. Implement the data processing flow.
2. Extendable in the future by configuring the current software or add functionality to the software.

The package that is created is called "**Botshot**" and consists of two software programs, see Table 8. The first part is creating the features for training from the datasets and is written in Java. The second part is designed with Python and does the training and validation.

Program	Language	Reason
Botshot Java	Java	A large amount of data should be processed (360GB). Even in Java, this can take a whole day. An interpreted language like Python is too slow. (Aruoba et al. 2014). With Java it is easier to make reusable code because of its Object Orientated approach.
Botshot Python	Python	The python language has a powerful syntax that is easy to use and simple to learn. Python require lesser code and is easy to debug. A lot of machine learning algorithms are available as libraries. The switch from one algorithm to another is easy. It is easy to build small programs that can be distributed as source code. With the reduced size of the feature set Python is the right choice.

Table 8: Botshot programs

5.1. Botshot Java

The Botshot Java program is used in the first three steps of the data processing flow. It is designed to be extendable and configurable. It can be used to handle the different scenarios that are needed to create a feature file for training. Table 9 and Table 10 gives a list of requirements that should be met.

Functional Requirement	
RQ1	Process the datasets and create a labelled flow. The datasets all have pcap files as input, but the truth is labelled can be different.
RQ2	Configure the program to use the dataset and how it should handle the truth.
RQ3	Make the program easily extendable to process new datasets with different truth labelling. If the code should be changed, it should be easy to do
RQ4	Processing a large amount of data can be time-consuming. Select the right programming language so it will not be to slow.
RQ5	Output statistical results about the processed data

Table 9: Botshot functional Requirements

Non-Functional Requirement	
RQ6	Use online versioning system to be able to distribute the software.
RQ7	Create documentation about the software, so it will be easy to do modifications
RQ8	Specify the minimum hardware requirements to run all datasets.

Table 10: Botshot Non-Functional Requirements

5.1.1. Design

The design of the Botshot Java software is built around the idea that there are different tasks that need to be done in sequence on a data structure. The sequence of tasks is called a job and can be programmed with a configuration file. The data structures where the tasks act on are called a flow. This gives us three main classes where other classes can be derived from.

1. IFlow: The data structure that is created when there is a new network flow created A new flow is a collection of network packets that belong to each other. When a flow is ready it can be processed by IFlowProcessor.
2. IFlowProcessor: This class processes a flow that is ready. It can add extra information to the flow, like truth value or output a file to store the flow in a file.
3. IWorker: This class creates the flows. Depending on the type of input, this can have different implementations. When a PCAP file is used, the flow should be created from the different network packets. But when an input already contains flows the information can be passed directly to the IFlowProcessor.

The Botshot Java program starts with the main function in CBotshot class see Figure 12. This function initializes three factories for the described classes above. See Appendix B for the complete class diagram. The factories are initialized with the different types of configurable classes that exist and can be used in the configuration file of Botshot java. The configuration file that Botshot java accepts as input is processed by the CWorkListCreator class. This class creates a list of jobs that should be handled. This list of jobs is of class CWorkItem and are the input for the CWorker class that coordinates all the work that needs to be done.

When the factories and worker class are initialized, the worker class starts to process all jobs. For every job, an IWorker class implementation is created by a factory. This class is used as an input class and reads the file to process. This can be the pcap file from one of the datasets. When the IWorker class implementation is initialized the process-function is called. The first thing a IWorker class implementation is doing is create an implementation for the IFlow class and creates a list of implementations from type IFlowProcessor. In the process-function the IWorker implementation is reading data to create a flow. When a flow is ready it calls the process-function on the list of IFlowProcessor implementations with the finished flow as parameter. When IWorker class implementation is ready with reading the input, it calls the finalize-function on the list of IFlowProcessor implementations.

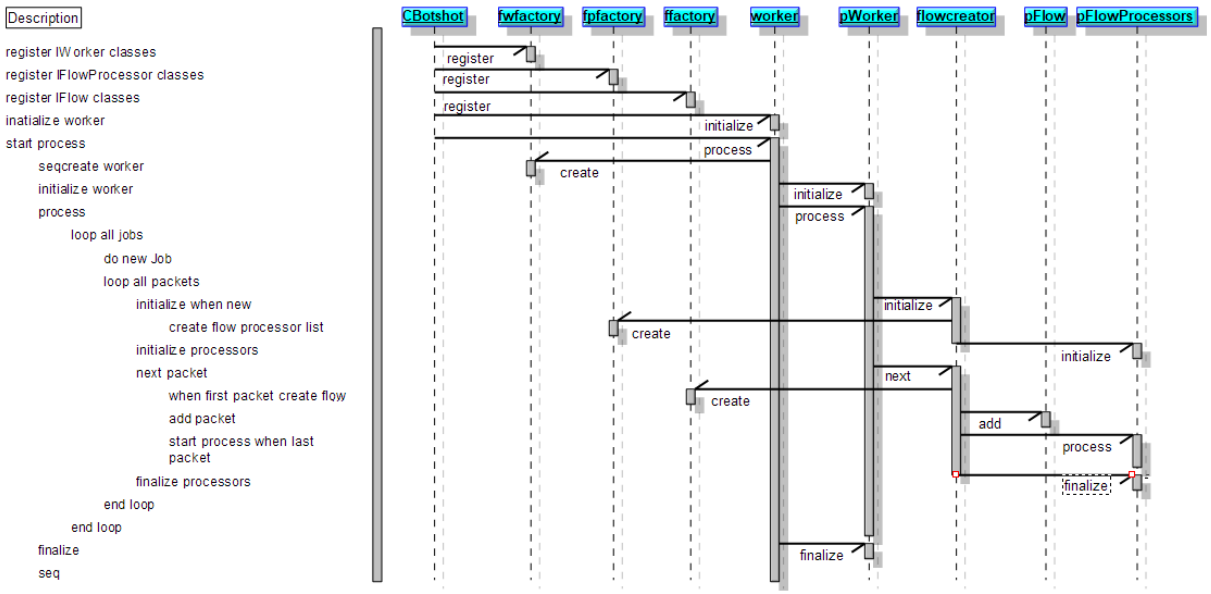


Figure 12: Botshot sequence diagram

To make it possible to configure the different created classes from a global configuration file, properties are passed to every instance of a class. The properties are used as an input to initialize the class. From a CWorkItem the function getInputAttributes returns a list of string pairs with the name and value of a property. In this way, the global configuration file can be used to supply initialization values to each class. Helper function of CWorkItem, contains setValue and getValue, make it easy to evaluate the properties. A code example is:

```
List<Pair<String,String>> attributes = work.getInputAttributes();

if (CWorkItem.contains(attributes,"file")) {
pcapFiles =
IWorker.getFileList(CWorkItem.getValue(attributes,"file").trim());
}
```

5.1.2. Implementation

The above description of Botshot Java is missing the different concrete implementation of the three abstract classes, IWorker, IFlowProcessor and IFlow. These abstract classes can be used to implement the concrete classes that do the actual work. For each abstract class, different concrete classes are created. Each concrete class is described in Appendix C, with the configuration parameters that it uses as input. The input parameters are called attributes and can be configured in the global configuration file.

5.1.3. Extension

Due to the flexible design of the Botshot Java program, it is easy to extend its functionality. By implementing one of the three interface classes (IWorker, IFlowProcessor or IFlow) new type of datasets can be processed. When a new dataset becomes available, it should be possible to create new training and validation features that can be combined with the current datasets. This makes it

easy to repeat this research or use the Botshot software for new datasets or algorithms.

5.1.4. Configuration

An important property of the Botshot java program is the ability to configure the functionality with a configuration file. The configuration file is in XML format and the location of the file is given as the first parameter of the Botshot java program. In the configuration file, different jobs can be defined that are executed by the program. The general structure is as follow

```
<?xml version="1.0"?>
<jobs>
  <job name="job name">
    <input name="input type name">
    </input>
    <flow name="flow name">
    </flow>
    <processor name="processor 1 name">
    </processor>
    <processor name="processor 2 name">
    </processor>
  </job>
 </jobs>
```

The job elements can be repeated to create multiple jobs to be executed. Every job can be given a name. The "job name" in the above example. This name is not used in processing but can be used when a file or log is written. A job consists of three child elements: an input, a flow, and a list of processors. The input element defines the input type and specifies a class in a factory of type "IWorker". The flow element defines the flow type and specifies a class in a factory of type "IFlow". At last, there is the processor element that specifies processors of type "IFlowProcessor". The list of processors is executed in the same way as they are defined in the configuration. This makes it possible to first execute a processor that adds the truth to a flow and then executes a processor that writes the flow to a file.

Every element can have extra child's as arguments. Below an example:

```
<flow name="Simple">
   <arguments>
     <argument name="timeout">600</argument>
     <argument name="skipport">53,138,137,123</argument>
   </arguments>
</flow>
```

This way of adding arguments can be done with all three job elements. When arguments are unknown, they are skipped. The different jobs are executed independently of each other. There is no functionality that results from one job can be passed to another job. The current implementation is running the jobs in sequence. To speed up the processing, this can also be done parallel, using the multiple cores of modern computers.

5.1.5. Run environment

An essential part of the Botshot Java program is the setup of the Java environment. The size of the data to process is extensive, and therefore the Java

runtime requires a lot of memory. The default java runtime settings are insufficient to run the Botshot Java program. With some extra settings, it is possible to tweak the Java runtime to be able to run the Botshot program. The following Table 11 lists the needed Java Virtual Machine options.

Option	Description
d64	Run java in 64 bit mode
Xmx8G	Specify the maximum memory to use. Default is 4G
XX:+UseParallelGC	Use a parallel garbage collector to speed-up the release of memory
XX:ParallelGCThreads=4	Use a maximum of 4 threads for the garbage collection
Djava.library.path= \jnetpcap-1.4.r1425	Specify the location of the jnetpcap library. This is to specify the location of the dll files in windows

Table 11: Java Virtual Machine options

These settings are found by looking at the java specifications (Oracle 2020). The use of the settings makes sure the program has enough memory and is fast. Not setting the correct parameters make the program slower or even unable to run.

5.1.6. Results

With the Botshot java program, it is possible to convert pcap files to network flow files with a truth value for further processing. The conversion from a pcap file to a feature file can be done in one pass. Because the processing of the files can take a lot of time, it is more convenient to output intermediate results that can be processed further. A second reason to first create the flow files and then the feature files separately, is that there is no option to modify the type of features in the file. When the process is split, it is easier to change the way features are calculated and regenerate the feature file. The time needed to create a feature file from a flow file is only a couple of minutes, depending on the speed of the computer.

The format of the intermediate flow file is defined in the class CFlow_Simple. The CFlowProcessor_Exporter class outputs a comma-separated file with the following format, Table 12.

Index	Name	Description
0	Flow type	What kind of flow is this TCP or UDP
1	Start time	The time the flow starts
2	Stop time	The time the flow stops
3	Source	IP source address
4	Source Port	IP source port
5	Source Cnt	Number of IP packets send
6	Source len	Total length of data sends
7	Destination	IP destination address
8	Dst Port	IP destination port
9	Dst Cnt	Number of IP packets received
10	Dst len	Total length of data received
11	Truth	0 if normal traffic, 1 for botnet traffic

12	Packet Cnt	Total number of network packets. After this value information of the different network packets is saved. The format is shown in the next table
	Packet info	Information about every packet
Last	End	End of flow with the string value "End"

Table 12: Intermediate flow format

Table 13 shows the values for the different network packets. The Packet cnt on index nr 12 gives the number of network packets that are saved before the End value. The character N is an index that can be calculated by multiplying the network packet number times the number of items per network packet.

Index	Name	Description
N+0	SorR	Is this network packet a send or received packet. Value is a string "SEND" / "RECEIVE"
N+1	Time	Time the network packet is send or received
N+2	Seq nr	The sequence number of the network packet
N+3	Ack	Is the ACK bit set "True" / "False"
N+4	Urg	Is the URG bit set "True" / "False"
N+5	Rst	Is the RST bit set "True" / "False"
N+6	Fin	Is the FIN bit set "True" / "False"
N+7	Psh	Is the PSH bit set "True" / "False"
N+8	Sync	Is the Sync bit set "True" / "False"
N+9	TtHop	HOP count in the network packet
N+10	TosTc	TOS from IP header
N+11	Size	Size of the network packet
N+12	Fragmented	Is the network packet fragmented in multiple packets?

Table 13: Network packet features

Every flow is ended with the string "End" and a carriage return. This makes it easy to read the lines. Because not every flow has the same number of network packets the number of values to read differs per line.

The conversion from a comma-separated flow file to a feature file, is done with a job that uses the class CWorkerCSV as an input class and the class CFlow_Sets as the flow class. A class CFlowProcessor_Exporter is used as a processor to output a CSV file with features. What types of features are saved is defined hardcoded in the software but is not fixed during the research. Specifying the file format is done when the features are discussed in chapter 6.

5.1.7. Improvements

The speed of the program is important. During development, multiple runs are done to make sure everything is working correctly. In the first iterations of the program, it would take multiple days to process all data sets. This makes it difficult to do multiple runs in a short time to make sure there are no problems with the program. The creation of the flows is taking up the most time. Looking up if a network packet belongs to a flow can take a lot of time. If a network packet belongs to a flow, the source address, destination address, and port numbers should be equal. The easiest way to do this is with al list structure. The code is easy to understand and easy to make. But when the list gets large, it will take a lot of time to look up the correct flow. Especially with UDP packets, it is

difficult to know when a flow ends, and the list gets long. This research is only using TCP packets but the botshot program is capable to also process UDP packets. By processing both TCP and UDP packets the botshot program is more usable for others, to research botnets network flows.

The first speedup is to discard UPD packets that are not part of the botnet traffic. Examples are DNS (port 53) packets, NETBios (port 137, 138) and NTP (port 123) packets. For the ISCX-IDS-2012 dataset 23% of the network packets can be discarded when these UDP packets are filtered out. Even with these optimizations, the time to process the pcap files is considerably large. The second optimization is to replace the list with a hash map. Java is very efficient in searching for the correct index. By creating an index from a pair with the source and destination address, the search for the correct index is much faster. Every map entry has a list of flows where only the port numbers are varying. Because this list is very short, a lookup for the correct entry is fast. The prototype for this map is:

```
Map<Pair<InetAddress,InetAddress>,List<IFlow>> mFlow = new HashMap<>();
```

This method is 4-5 times faster than only a list. The downside is that the code is a little more complicated and less easy to understand and change.

Even with this method to improve the speed of the program, it can take multiple hours to process a dataset. Profiling the program shows that the creation of objects, by the factories, is taking a long time. For most objects, this is not a problem, but for the flow objects, this is different. Flow objects are created often and are delaying the program. A solution can be to create a pool of flow objects. By not deleting the used flows but store them in an object pool and reuse the already created flows, the program speed can be 30% faster. The downside is that the program uses the maximum amount of memory for a longer time, till the end of the program. When you execute the program multiple times to process different data set in parallel, the memory usage can be a problem. This can be solved by adding more memory to the computer.

Following a TCP flow can be difficult. Usually, the TCP connection is fault-tolerant, and packets can be resent. This requires a complex state machine to follow what is happening. To make the program easier to read and program, this retransmits of packets is discarded. We assume there are no faults in the transmission of packets. A second simplification is that we modify TCP packets that are built from multiple IP packets. In this case we store only the first packets and discard the rest. The calculated packet length is then wrong. Because there only a couple of network packets that have multiple IP network packets in a TCP packet it will not influence the results.

5.2. Botshot Python

In the previous paragraph, the Botshot-java program is introduced. One of the files this program can produce is a feature file that can be used for training and evaluation. The feature files contain a feature-vector for each flow that is created from the datasets. A feature file can be used for training a model. To train a model and produce validation results four little python program are created, see Table 14. The programs are little one-page programs that are easy to understand and modified. The choice to make four small programs instead of one larger program is made because of its simplicity. Every program has a dedicated task and is easy to debug and adapt to new scenarios. One big program would mean

reading configuration files etc. Better if you want to automate the process but too complex to do research.

Nr	Program name	Description
1	Botshot-Balance	Program to balance the data. The program takes a feature file as an input and output a sub-sampled and over-sampled feature file. This program can modify the feature-vector if needed. For example, to throw away features that are not needed. The feature file written to disk contains only the features needed and has balanced the two classes.
2	Botshot-Train	A program that uses a feature file as input. Normally this would be the balanced feature file from the previous program but can also be the unbalanced file directly from Botshot-java. The program uses this feature file to train a model for one of the selected algorithms. The model is written to disk and can be used later to validate the results. Many log files and graphics are created from the training results to evaluate the created model visually.
3	Botshot-optimize	Program to optimize the number of features used to train a model. This program is an extended version of the Botshot-Train program. In a loop the number of features is reduced to find the smallest feature-vector that give good results.
4	Botshot-validate	A program that can use a trained model to validate a feature-set. This last program used a trained model from the Botshot-Train program to evaluate a feature-set and output results. The feature-set can be created by Botshot-java or balanced by Botshot-Balance.

Table 14: Botshot python programs

The four programs all have the same overall structure, see Figure 13. The first section holds the settings used in the program. Settings include the file names for input and output etc. Then there is a section that initialize the program and read the files. When everything is initialized, the execute section is for the programs real work. Examples are balancing the data or training a model. The last sections produce the results. The results can be shown as text or graphics or an output to disk. Most information is logged to a file for later inspection, and a lot of graphics can also be stored on disk.

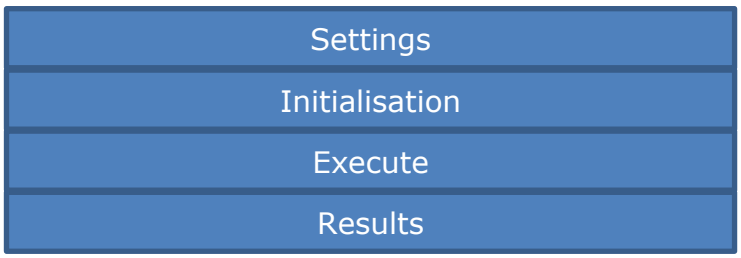


Figure 13: Botshot python program structure

For the training of the model, using the feature files, the Scikit-learn library is used. That library is a python library designed for evaluating machine

learning algorithms. With the use of this library and python, it is easy to do experiments quick. To make debugging easier and have a simple method to output intermediate results the python programs are designed with the use of a "jupyter notebook" (www.jupyter.org 2020). A jupyter notebook makes it easy to create small python programs that stores intermediate results. The program can be created step by step without running the complete program again. Because some parts can take a long time to execute the storage of intermediate results can save a lot of time when parts of a program are designed or debugged. Using this feature of a jupyter notebook correctly, requires some changes to the programming style. You need to make sure you are not modifying variables that are not declared in the section you are working with. When variables are changed from previous sections the re-execution of the current section starts with different, modified, input variable. This becomes clear in the programs when input variables are copied before they are modified.
The four small python programs can be viewed in Appendix D.

6. Experiments

With the results from experiments an answer can be given on the research questions. In the next section the sub-research questions will be used to introduce the experiments needed to give an answer. A total of 5 experiments are done, see Table 18.

Experiment	Description
1	Train algorithms with a feature-vector inspired on the work of van Klaveren (van Klaveren 2019)
2	Improve the Feature-vector
3	New dataset with improved feature-vector
4	Evaluate new dataset with previous trained model
5	Evaluate effectiveness of the algorithms

Table 15: Experiments

6.1. Introduction

With the created programs, experiments are done to answer the research sub-questions. The first research question RQ1 is answered in chapter 4. With the selected datasets, feature-sets can be created for training and validation. A feature-set contains multiple feature-vectors with a truth value. A feature-vector represents one network flow.

The botshot-java program is used to convert the datasets into three feature-sets. See Table 16. Each feature-set is created for a different reason.

Feature-set	Dataset	Purpose
1	PeerRush, ISCX IDS 2012	These are the two largest datasets. That creates a good mix of regular network traffic and traffic from two usable botnets: Waledac and IRC Botnet. The Storm botnet is using UDP messages which are discarded.
2	PeerRush, ISCX IDS 2012, ISOT, CTU 13	Extended feature-set with data from the ISOT and CTU 13 dataset. In the ISOT dataset contains examples of regular network traffic and Waledac botnet. CTU 13 extends the number of botnets with Neris, RBot, Virut, Menti, Soguo, Murlo and NSIS.ay
3	ISCX Bot 2014	This feature-set is created with data from the ISCX Bot 2014 dataset. This dataset is partly overlapping the other datasets but also has some new botnets, see Appendix A. The network traffic from regular programs is also present in the other datasets.

Table 16: Feature-sets for experiments

In chapter 3 the answer is given on which machine-learning algorithms are selected. We applied three machine-learning algorithms for comparison. All experiments are done with those three machine-learning algorithms to find the best result. The algorithms used are listed in Table 17.

Nr	Algorithm
1	Support Vector Machine (SVM)
2	Random Forrest (RFC)
3	Gradient Boosted Trees (GBT)

Table 17: Machine-learning algorithms

The first experiment is to evaluate the performance of the classifiers with a feature-set number one. This feature-set is used because it contains only two types of botnets. With a smaller number of botnets, the classification will probably be easier. In later experiments the complexity is increased.

In the second experiment, the feature-vector size is reduced. What is the smallest feature-vector where the accuracy is not dropping? A smaller feature-vector means a less complicated model, optimizing the bias-variance trade-off. In general a less complex model will be more general making it better suited for evaluating new data. Smaller feature-vectors also reduces the training and evaluation time making it faster to do experiments. With this feature-vector, the second research question is answered. For this experiment also feature-set one is used.

In the third experiment, feature-set two is used. This feature-set contains more network traffic from new botnets and extra regular network traffic. This creates a more complex problem for the algorithms.

In experiment four a feature-set is used that contains new botnet network traffic. In this experiment, the performance of the model is evaluated when new traffic type is added to the feature-set. The used feature-set, for this experiment, is number three.

With the last experiment the performance question is answered. The relative speed performance of all the different algorithms is calculated.

In all experiments, the feature-sets are balanced. This is done to be able to compare the results with each other. For the comparison between the experiments, the accuracy is used. Since the accuracy is sensitive to data in-balance, balancing the data is done with two modes, under-sampling, and over-sampling. With under-sampling data is removed from the data set and with over-sampling data is duplicated. Both methods can influence the outcome of the accuracy value. If the accuracy values from both ways are close, we can assume the accuracy is valid for the future-set.

6.2. Experiment 1: Features

In this experiment, a model is learned for each machine learning algorithm that is selected. An algorithm can only be trained if a feature-set is available with feature-values. As a start, the work from Roosmalen (van Roosmalen 2017) and van Klaveren (van Klaveren 2019) is used to create a feature-vector. Both algorithms from van Roosmalen and van Klaveren use a massive number of feature-vectors for training their algorithm. Van Roosmalen uses a 4158 features wide feature-vector, and van Klaveren uses a feature-vector size of 320 features to get good results. These feature-vectors are too large to be used with Scikit-learn. The feature-set should fit entirely inside the memory of the computer, with these extensive feature-vectors that is not possible.

The solution is to use a smaller feature-vector for training. A lot of features can be combined because they are the same for all network packets inside a flow. The work from van Klaveren is used to create a selection of

features that are important. For the first experiment, we use the feature-vector from Table 18. This are features extracted from each network flow.

Nr	Feature	Remark
1	duration	The duration of the flow. The time of the last network package minus the time of the first network package.
2	src port	The source port of all the network packets
3	dst port	The destination port of all the network packets.
4	pack_s	The number of network packets send
5	send	The total number of bytes send
6	pack_r	The number of network packets received
7	received	The total number of bytes received
8	Rst	Is the flow ended normally or ended with a timeout? Flows that have not send or received a package for 6min. are ended without a reset packet.
9	S0	The size of the first network package. If the number is positive, it is a package sent. Is the number negative the package is received
...	…	The size of the network package second up to sixty-fourth
73	S63	The size of the 63rd network package. If the number is positive, it is a package sent. Is the number negative the package is received

Table 18: Features

As stated earlier, all features should be independent of the network topology used. This to ensure the trained model will be usable in a different network environment. Features like the TCP/IP address are particular for a network. In a normal situation, they can be different. Also, the TTL value from the TCP/IP header is too specific. It shows the number of hops from and to the destination computer. Because van Klaveren uses TTL in the feature-vector and it looks important, we have tried to see if it is also important for the three algorithms used. Adding the TLL into the feature-vector did not improve the results. We will not use the TLL feature in our experiments. A reason that TTL has some importance can be that this value can separate the different datasets from each other. For example, in the ISOT dataset the TLL will be small and equal for all network packets. This because the dataset is created by replaying network packets in a small, controlled network environment. When the datasets are distinguishable for the model it reduces complexity for classification.

The feature-vector contains two types of data; the first eight features are global flow features. The rest are features from the individual network packets. We use the same features as van Klaveren, only where features are the same in all network packets, we use one feature, called a global feature. Using the same feature multiple times does not add any information. Because we do not use the values from all network packets, some features represent all packets. The packet size and number of packets are such features.

The first packet inside a flow is called the sent packet and defines the direction. Network packets that have the same source and destination address as the first packet are always stored as sent packet. Network packets the have the source and destination address swapped according to the first packet are stored as received packets. Sent packets uses a positive payload size and received packets use a negative payload size. This is an optimisation from the features

used by van Klaveren, where an extra feature is encoding the direction of the network package.

From the TCP/IP structure, not all features are used. Table 19 give a list of features that are not used and why they are not used.

Name	Description	Why
IP	IP version	Value is always constant
IHL	Size of the IP Header	Value is always constant
DSCP/TOS	Type of service	Easily changed by a program
ECN	Explicit Congestion	Value is constant
Fragment	Fragment Offset	We use the flow, do not need to reconstruct the data
TTL	Time to Live	Is reflecting the network topology
Protocol	Protocol Number	Value is always constant
Checksum	Header and data checksums	Dependents on the payload data which we discard
IP-flags	Three 1-bit flags	Controls fragmentation of IP packets. That is discarded.
IP-Options	Options field	Not often used. We discard the options.
Sequence	Sequence number	We store the network packets from a flow in a sequence. This number is not needed
Flags control bits	Nine 1-bit flags	Control the flow of the TCP packet. Is used during flow creation
Payload	Data of the package	The payload of the messages is often encrypted or can be easily encrypted

Table 19: Unused Features

The first step after selecting the features is to balance the training data. For all training cycles, we will make sure the number of feature-vectors is the same for both classes. There are multiple technologies to balance the data. For these experiments, we use only one type "sub-sampling". Sub-sampling will reduce the number of feature-vectors. This is needed to make sure all data will fit inside memory before training starts. Other methods of balancing the data will increase the number of feature-vectors.

The reduction of the feature-set can make the feature-set not representative for the complete set. To make sure the reduced feature-set is representative for the full set, the training can be done on multiple reduced feature-sets. The average accuracy result of these trainings will give the final accuracy results.

Results

The balanced feature-set is split into a training and validation set. 75% of the feature-vectors are used for training, and 25% are used for validation.

The results from training and validation for the first feature-set are as shown in Table 20: Validation results experiment 1.

Property	SVM	RFC	GBT
Sample size	3,814,510	3,814,510	3,814,510
Training time	119s	260s	12720s
Validation time	1.52s	5.42s	12.31s
Accuracy %	82.42	99.22	97.40
MCC %	64.16	98.45	94.80

Table 20: Validation results experiment 1

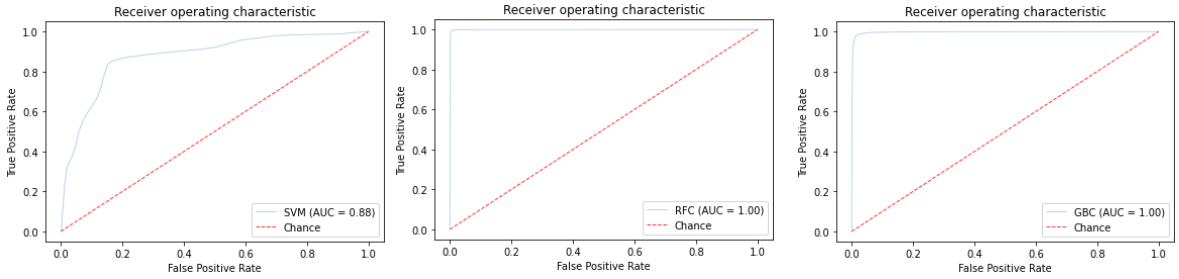


Figure 14: ROC curves

The ROC curves from the three different trained models are shown in Figure 10. The curve from the random forest classifier (RFC) has the sharpest corner. This indicates this is the best classifier.

The SVM trainer uses a linear kernel. This is probably not the optimal kernel, but other kernels did not complete. Multiple kernels are tried, but none of these kernels finished training after 48 hours of running. What the optimal kernel will be is difficult to predict in advance and should be tried. Maybe with smaller datasets it is possible to train an SVM with different kernels. A reduction of a factor 4 is tried but still the training time is too large. Randomly reducing the dataset even more can be done but the question is if the results from the classifiers can still be compared. In this research there is no time spent in using smarter ways to reduce a dataset.

The results from the RFC algorithm are the best from the three tried algorithms. The accuracy of 99.22% indicates that the used feature-vector contains the right information to separate the botnet flows from the regular network flows.

From the used algorithms, it is easy to get an importance value per feature, what can be used in experiment 2. This can help to reduce the feature-vector size. A smaller feature-vector will improve training time, evaluation time, the memory needed, bias and variance.

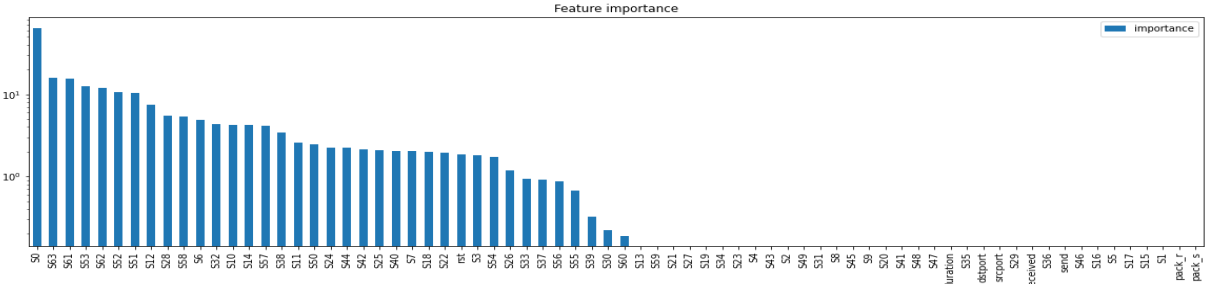


Figure 15: Feature importance SVM

The feature importance list of the SVM, Figure 15, is selected by using the SVM coefficients from the training. This works only for linear kernels because the inputs and weights are in the same dimensions. When non-linear kernels are used the weights are transformed to another "dimension", and this does trick does not work (Guyon et al. 2003).

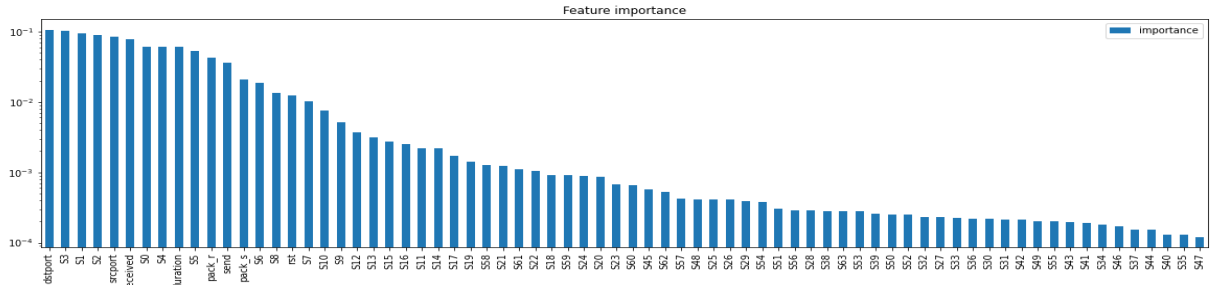


Figure 16: Feature importance RFC

The feature importance for the RFC is based on the Gini index. When computing a single tree, it is possible to calculate how each feature contributes to the result. The combination of all the trees gives the feature importance for one feature. Figure 16 gives the feature importance for the RFC for each feature in the feature-vector. Because the training-set is used to evaluate the feature importance, this method is sensitive for biases and make a feature that is continues more important than discrete features. This method is fast and easy to use. A better approach would be to use permutation importance for feature evaluation (Breiman 2001). This method uses the validation set to calculate the feature importance and does not have the same problems. Due to the large dataset, this method takes too long to calculate to be useful.

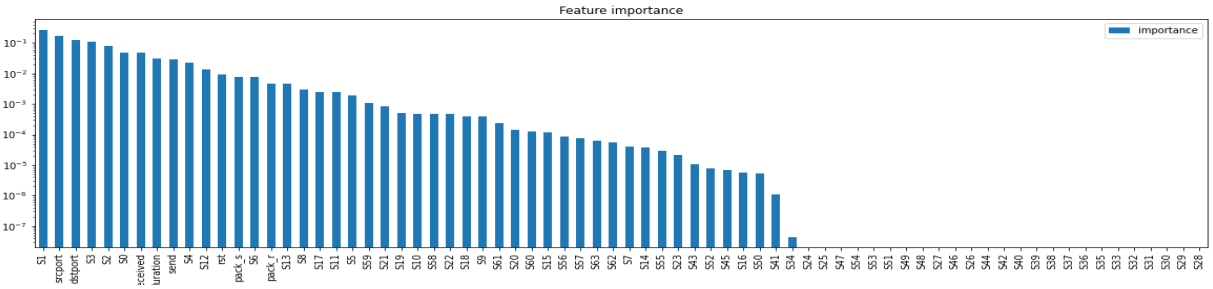


Figure 17: Feature importance GBT

The feature importance for Gradient Booted Trees is calculated similarly to the Random Forest. This has the same bias problems and the same solution using the permutation importance for feature evaluation (Breiman 2001). But again, the size of the dataset is a problem with this method and therefore not used. The feature importance per feature is shown in Figure 17.

6.3. Experiment 2: Improve the feature-vector

Looking at the results of experiment 1 there is some room for improvement. In experiment 2 the feature-vector is evaluated. Are there features that do not contribute to the answer. To speed-up the generation of results, only the random forest classifier is used. It was the best classifier in the previous experiment, and it was fast in training and evaluation.

In experiment 1 a relatively small feature-vector is used, compared to van Roosmalen (van Roosmalen 2017), but from the feature importance list, it is already clear the size of the feature-vector can be reduced. As a first step, the

features "S16" till "S64" are removed and the accuracy calculated. Because the accuracy is not affected after the removal of the 48 feature values the remaining 24 feature values are used to look for the best features to use. A loop is created where every iteration the least important feature is removed. Each iteration a new feature important list is created, and the least important feature removed. This is repeated till there is only one feature left. Figure 18 gives the accuracy and MCC percentage of a new training after removal of the least important feature. This method is called greedy search with backward elimination. (Guyon et al. 2003)

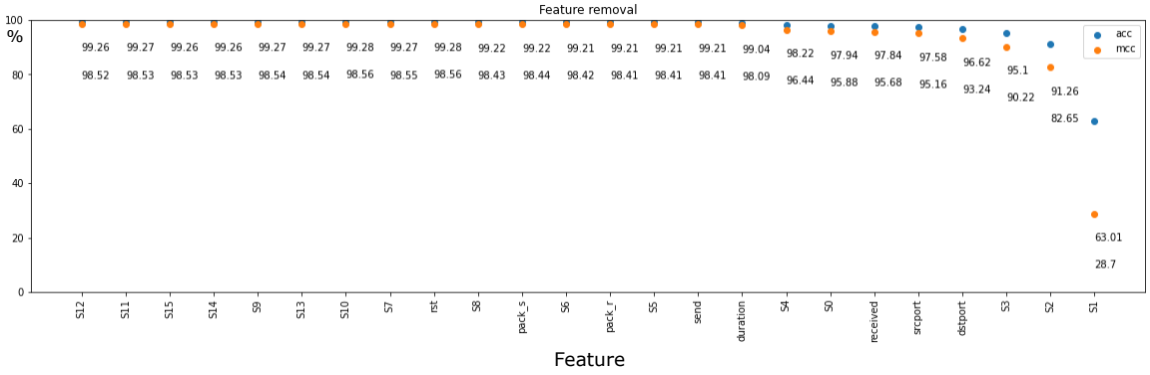


Figure 18: Feature removal

From Figure 18 it can be seen we need only 16 features before the accuracy and the MCC percentages are dropping to a lower percentage. This gives us the confidence we can continue with a much smaller feature-vector. Because S8 (payload size of network packet 8) is an important feature, we can assume S7 will also have some importance. In further experiments, we use a feature-vector size of 17 features. The eight global features added with the size features of the first nine network packets. The reduction of the feature-vector size results in a less complex model with the same performance.

Because the feature-vector is only 1/3 of the feature-vector we start with, in experiment 1, it will now be possible to use different balancing methods. In experiment 1, the sub- or under-sampling method is used. The question is, what will happen with the performance of the trained model if the sample-set is over-sampled. This will increase the sample size and uses all available data.

Balance	Sample size	Training time	Evaluation time	Accuracy %	MCC %
sub	3,814,510	260s	5.42s	99.22	98.45
over	11,715,180	895s	18.04s	99.63	99.36

Table 21: Validation results experiment 2

Using the over-sampled data, the accuracy increases to 99.63%

Random Forest classifiers are not sensitive for overfitting (Breiman 2001) and (Louppe 2015), but as for every machine learning algorithm, it can be overfitted in special cases. In the case, the Random Forest exactly fits the training data the Random Forest is overfitted and learned the data exactly. The solution would be to increase the dataset. Because we use a vast number of feature-vectors and only a small number of features, the chance of overfitting is minimal. The number of different feature-vectors is much larger than the number of features in one vector. (Briscoe et al. 2011)

The reduced feature-vector is not using all the information from the network packets. Especially the sizes of the packets later in the flow are discarded. Maybe there is more info stored inside the size values. The following things are tried.

1. Adding the average and standard deviations of the packet sizes.
2. Adding a size histogram with fixed bin sizes of 50 bytes.
3. Adding a size histogram with logarithmic bin sizes. Using formula Eq 14

$$x = {}^{10}log\left((\boldsymbol{MAX\_PACKET\_LENGTH}) * 10\right) + 0.5) \quad \text{Eq 14}$$

The average and standard deviations are also used by Wang et al. (Wang et al. 2020) in their feature vector. The creation of the histograms is an attempt to capture the sizes of the network packages in a flow in a fixed number of values. The bin size of 50 is an arbitrary value that gives with 30 features when the maximum transmission unit is used from ethernet packages (Kozierok 2005). Because the number of network packets with smaller sizes is much larger than with larger sizes and with smaller size a smaller bin size can give more information the formula from Eq 14 is used to make logarithmic bin sizes, from small to large. With maximum transmission unit from ethernet packages (1500) the histogram size will be 32 values.

These features have been added to the 17 important features and models are trained. After evaluation of the importance of these features, none of these features has a high score. These features do not add any information that improves the accuracy of the model.

6.4. Experiment 3: New model with more data

The above experiments are done with a dataset that has a limited number of botnet and regular network traffic. The second created feature-set has more botnet types and normal network traffic. With a feature-vector of 17 values, we can make a model. Because of the size of the feature-set, only the Random Forest Classifier is used.

Balance	Sample size	Training time	Evaluation time	Accuracy %	MCC %
Sub	3,814,510	250s	4.72s	98.86	97.73
Over	13,309,792	1000s	18.27s	99.57	99.14

Table 22: Validation results experiment 3

The performance of the random forest model is performing a little bit less with this extended feature-set. It has more problems classifying the data. This could be due to the reduced feature-vector. The large 72-value feature-vector from experiment 1, is also tried but did not give a better result. The feature-vector with 17 values is again valid for this feature-set.

6.5. Experiment 4: Validate with unknown flows

With this experiment, we want to see how good the model performs with new flows from unseen botnet traffic for which the classifier has not been trained. This will be done in two steps. First, we use the model created in experiment 2 with the over-sampled data and use the data from the second feature-set. This

will give an idea of how good the model performs on new network traffic. Most new network traffic comes from regular traffic; only a small part is coming from new botnets. Secondly, we can test the model with a completely new feature-set that is mostly different than the feature-set used for training.

To test the performance of the random forest model we use the sub- and over-sampled feature-set two with the model from experiment 2.

Balance	Sample size	Evaluation time	Accuracy %	MCC %
Sub	3,814,510	4.72s	96.86	93.72
Over	13,309,792	18.27s	96.90	93.79

Table 23: Validation results experiment 4 model from experiment 2

From the results, it is visible that the performance of the model is decreasing when unseen flows are evaluated.

We can do the same with the third feature-set that is defined, the ISCX Bot 2014 dataset with mostly unseen data.

Balance	Sample size	Evaluation time	Accuracy %	MCC %
Sub	153,476	0.56s	57.32	25.13
Over	385,502	1.26s	57.31	25.11

Table 24: Validation results experiment 4 new data

The results show that the random forest model is not working with new features-sets. The model works with features that look like the flows that are in the training set but with new feature-sets it fails.

	Sub-samples		
		True	False
Truth	True	9%	41%
	False	2%	48%
		Prediction	

	Over-sampled		
		True	False
Truth	True	9%	41%
	False	2%	48%
		Prediction	

Table 25: Validation results experiment 4 Confusion matrix

From the confusion matrixes above it is visible that the detection of botnets is very low. From the 50% flows from botnets, only 9% is detected. The detection of regular network flows is a lot higher. That is because most regular network flows are also present in the training set.

The Random Forest Classifier was performing the best in the first experiment on known data. It could be that SVM or GBT are doing better with new data. Using the over-sampled feature-set from experiment 2, the following results are acquired, see table Table 26.

Type	Sample size	Evaluation time	Accuracy %	MCC %
SVM	385,502	0.17s	59.54	19.21
GBT	385,502	1.23s	55.59	15.72

Table 26: Validation results experiment 4 new data

The results are comparable with the results from the Random Forest Classifier. The classifiers are unable to classify a new feature-set correctly. An accuracy result of 50% means that only half of the samples are classified correctly. The three classifiers perform only a little bit better than randomly select the class.

It could be that this new feature-set requires different features than the feature-set used for training. What features will be important when the training is done with feature-set three. The result from the training with an oversamples feature-set are shown in Table 27.

Type	Sample size	Training time	Accuracy %	MCC %
RFC	385,502	19.84s	99.33	98.66

Table 27: Validation results experiment 4 with new model

The results show that the RFC can detect botnets in network flows from feature-set three. Figure 19 shows that the same features are important as in experiment one and two.

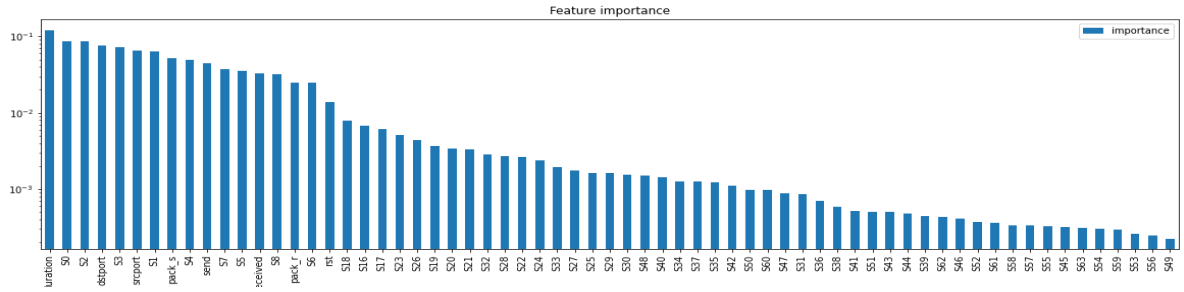


Figure 19: RFC feature importance for feature-set three

Because the important features are the same from earlier experiments, we know we have selected the correct features to do the classification.

6.6. Experiment 5: Computer performance

All previous experiments are done on a single computer. To be able to compare the different algorithms on their effectiveness, the accuracy value is used with a balanced test set. In the last experiment, we determine the speed of the computer where all experiments were done. With the speed of the computer, the time needed to train a model and evaluate a feature-vector can be placed in perspective. New algorithms can not only be compared with their accuracy but also with the resources needed to do training or an evaluation.

The speed of the computer is given in GFlops and calculated with Linpack package from Intel (Fedorov, Zhu and Singh 2020). The Intel package reports a computer speed depending on the calculated problem. From the different speed calculations, in the Linpack package, the average is taken.

CPU	Memory size	GFlops
Intel Xeon E3-1245 V3 @ 3.40GHz	16Gb	134

Table 28: Computer speed

Using the equations from section 3 the effectiveness of the algorithms can be calculated using the results from experiment 1

Type	eff_t	eff_e Training	eff_e Evaluation
SVM	100	239	18727

RFC	218	109	5252
GBT	10689	2.2	2312

Table 29: Algorithm efficiency

The calculated effectiveness gives a relative number independent of the used computer. The higher the number the more effective the implemented algorithm uses the resources of the computer. A remark is that the number is dependent on how the algorithm is implemented. In Scikit-learn the SVM and GBT algorithms are not optimized to use all cores of the processor. Where the RFC algorithm is using all cores and is highly optimized, especially for evaluation, it should be easy to better use the full capacity of the processor. With better optimized implementations of the algorithms the number can be different.

6.7. Overview

In the first four experiments, different results are presented. Because it can be challenging to overview all the experiments, Table 30 is summing them all up.

Experiment	Feature-set	Algorithm	Acc %	MCC %
1	1 under-sampled	SVM	82.42	64.16
	1 under-sampled	RFC	99.22	98.45
	1 under-sampled	GBT	97.40	94.80
2	1 under-sampled	RFC	99.22	98.45
	1 over-sampled	RFC	**99.63**	**99.36**
3	2 under-sampled	RFC	98.86	97.73
	2 over-sampled	RFC	99.57	99.14
4	2 under-sampled	RFC	96.86	93.72
	2 over-sampled	RFC	96.90	93.79
	3 under-sampled	RFC	57.32	25.13
	3 over-sampled	RFC	57.31	25.11
	3 under-sampled	SVM	59.54	19.21
	3 under-sampled	GBT	55.59	15.72
	3 over-sampled	RFC	99.33	98.66

Table 30: Experiment results overview

7. Conclusion

In the previous chapters, we have tried to get an answer to the main research question by answering the four sub-questions.

7.1. Questions

For the first sub-question, RQ1, an answer is given in chapter 4.

RQ1: Which dataset will be used to train and evaluate machine learning algorithms?

In chapter 5, we have selected five datasets that are created by others. These datasets are used to create training- set and evaluation- sets. The five datasets are:

1. PeerRush
2. ISOT
3. ISCX IDS 2012
4. ISCX Bot 2014
5. CTU 13

In total these datasets contain 1,679,230,826 network packets that can be converted to 9,269,413 TCP/IP network flows. The ratio between botnet traffic and regular traffic is 1:3.4. The data is relatively old and contains data from botnets and applications that are not always active anymore. Not all data is entirely unique. Some data is directly shared between datasets, and some data is recreated using a different network topology. For creating a training set, this duplication is discarded. The datasets are especially created to do research on detecting botnets. Many researchers use one off or a combination of these data sets to test algorithms for detecting botnet network traffic. (Roosmalen et al. 2018; Pektaş et al. 2019; Zhao et al. 2013; Alauthaman et al. 2018).

The second question RQ2 is questioning the features to be used.

RQ2: Which training features, from TCP/IP network traffic, should be used for botnet detection?

The used features come into two categories. The first are flow characteristics and the second individual network packet characteristics. In Table 18, all features used are described. From the network packets, only the payload size is used. The rest of the features from the network packets are also available as a flow feature or are network topology dependent. During trunking of the feature-vector size, most network packet payload sizes are removed. Only the sizes of the first eight packets are essential. All global flow features are necessary to train an algorithm.

Because most payload size features are removed from the feature-vector the question could be asked if valuable information is not lost. Adding more features about the sizes of the network packets did not create a better model. During trunking of the feature-vectors, these features are always removed. The used algorithms did not use them.

In comparison with the feature size used by van Klaveren the number of features is reduced dramatically. In essence the used features are the same. A lot of features are globally for the network flow and do not have to be repeated

in all packets. Most features use the payload size as a basis. Because the global features tell a lot about the total payload sent and received not all individual network package payload feature have to be used. This reduces the feature size drastically. Most network flows have a limited number of packets inside, see Figure 20. This figure shows the percentage of flows that have a certain number of packets inside.

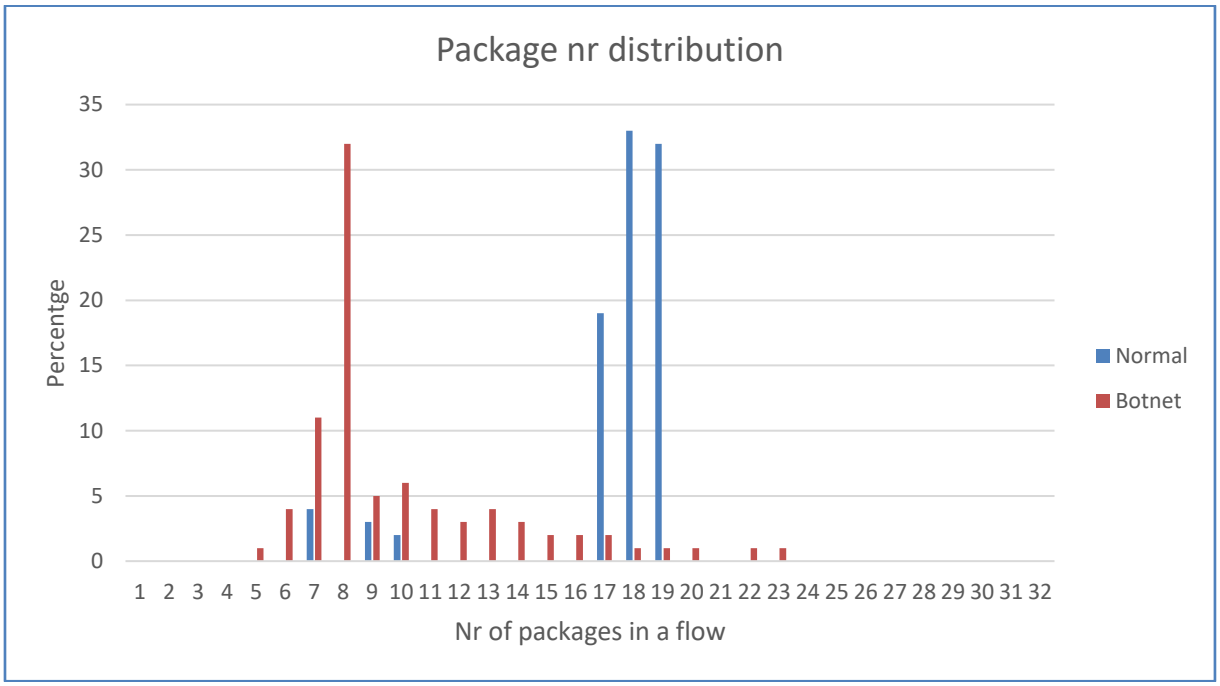


Figure 20 : Package nr distribution

Extending the feature-vector to include the flow with the most packets, as done by van Klaveren, is not necessary. The first nine sizes give enough information, as shown in experiment 2. Also experiment 4 confirms that the same features are important when a completely new data set is used for training. Giving confidence we use the correct features.

The third question is about machine learning algorithms to use. RQ3

RQ3: What selection of machine learning algorithms can be used best to detect botnets?

The answered is given in chapter 3. The following algorithms are used to detect botnets.

Nr	Algorithm
1	Support Vector Machine (SVM)
2	Random Forrest (RFC)
3	Gradient Boosted Trees (GBT)

Table 31: Used machine learning algorithms

From literature, these three machine-learning algorithms are performing best in most situations.

In our experiments, we could only use a linear kernel for the SVM. Other kernels take too long to train (> 48 hours). The linear kernel was not performing

very well, as can be seen in Table 20. The GBT algorithm is performing average but needs a lot of time to complete. For the rest of the experiments, only RFC is used because it performs best and trains in a short time.

The best result is achieved with feature-set one that is over-sampled. With this feature-set, the RFC algorithm gets an accuracy of 99.63%. The RFC algorithm is good at classifying data that it has seen during training but fails to classify data that is not in the dataset. In the experiment where the unseen data is used to evaluate the performance of the RFC algorithm, the accuracy is dropping to 57.31%. This is just a little better than using a random value algorithm. It shows clearly that the RFC algorithm is unable to classify new data that is not used during training. The used feature-vector is not specific enough to detect new botnet flows.

This makes it difficult to use this algorithm in a real application. Network traffic that is coming from new applications or new botnets will have a big chance of being classified incorrectly. This will create a lot of false positives and true negatives.

In experiment 3, the performance of the Random Forest Classifier is also dropping. Although the data used to train the RFC is also used for validation, the performance is lower. When the diversity in the two classes is growing, it is harder for the RFC to maintain its accuracy.

The last question to answer is RQ4

RQ4 How can the different algorithms be compared in their effectiveness and efficiency in detecting botnets?

In experiment 5, a relative number of efficiencies is calculated. The used algorithms can be compared with other algorithms but with some remarks. The used algorithms are optimized differently and use the available resources differently. Only for the current algorithm implementation, a conclusion can be given, but this is not telling if this is the best implementation. The calculated numbers can be used for comparison, but their use is limited.

The effectiveness of the algorithms is presented as the accuracy during validation. Because we balanced all feature-sets the different accuracy numbers can be compared. The accuracy is often given in other research papers to express their effectiveness. To really compare algorithms, the accuracy can only be a good value when the dataset are the same.

With the answers on the four sub-questions the main research question can be answered, the main research question is:

How can machine learning techniques effectively and efficiently detect botnets from TCP/IP network traffic?

During this research, we have shown that the Random Forest Classifier can detect botnet TCP/IP network traffic effectively and efficiently. We have shown how to implement this and how to create a dataset for training a model.

There are some remarks on how effective the Random Forest Classifier will be in a real application with a larger diversity of network traffic. Also, the age of the available dataset is questionable, are they still representative for modern network traffic.

7.2. Comparing results

The botshot program is generating good results on detecting botnets traffic with an accuracy of **99.63%**. The remark is that the botnet traffic should be part of the training data. The detection of new botnets is not accurate with the botshot program. More researchers have created algorithms that have very good results. Discuss them all is impossible, but some examples are.

Roosmalen et al. (Roosmalen et al. 2018)

The accuracy reported is **99.7%** on a balanced dataset using a large neural network. The network traffic is converted to flows and converted to input vectors for a neural network. The dataset contains TCP and UDP botnet traffic.

Van Klaveren (van Klaveren 2019)

The accuracy reported is **97%** on a balanced dataset. A large neural network is used but considerable smaller than the network used by Roosmalen. This work is proposing a smaller feature-vector by removing non important features as used by Roosmalen.

Wang et al. (Wang et al. 2020)

The accuracy reported is **99.94%** using a flow- and graph-based feature model. In total 18 features are used. Some features are not network topology independent. There some time features introduced that could be interesting.

Pektaş et al. (Pektaş et al. 2019)

By using a graph-based modelling technique a performance is reached of **99%** accuracy. The graph feature looks comparable to the global features used in our research. The method used is a convolutional neural network with a large input vector containing all flow features sequenced by a full connected neural network combining the CNN with the graph features.

Zhao et al. (Zhao et al. 2013)

The reported performance is a true positive rate of over **90%** and a false positive rate under **5%**. This is done using a flow- and graph-based method. The detector is tried on new data but is producing a high false positive rate of **82%**. The used method is not independent of the network topology.

Comparing results is difficult because the reported performance is depending on many variables. First the used datasets are not equal, and the dataset is not always balanced the same way.
The botshot program uses minimal number of network topology independent features and reports an accuracy close the best performing algorithms which sometimes using discussable features.

7.3. Discussion

In the experiments only a small feature-vector size is used with mainly network payload size features. These payload size features are absolute values. The

threshold inside the trained algorithms is based on these absolute values. Small changes in these values can have a huge effect on the performance of the algorithm. Maybe there is a method to make the size features more relative. Using the delta size from package to package or other mathematics to reduce the sensitivity of the used size features.

The flows to train the algorithms are all used without modification. This is a huge dataset with all kinds of data. The question could be asked if there is no repetitive data inside the dataset. Is it not possible to reduce the number of flows by combining the flows that are equal or almost equal? It can be that when there are large groups of equal flows inside the dataset the algorithms get biased toward these flows. This can influence the performance of the algorithms when new data is evaluated.

One method of finding equal flows can be the distance between two feature-vectors as a value of equalness. The distance between two feature-vectors can be calculated according to formula Eq 15.

$$S = \sqrt{\sum_{i=0}^{16} \left(f_i^x - f_i^y\right)^2} \qquad \text{Eq 15}$$

where:

$f_i^x = The\ feature\ value\ of\ feature\ vector\ x$

$f_i^y = The\ feature\ value\ of\ feature\ vector\ y$

In our experiments only the linear kernel of an the SVM classifier could be used because our dataset was too large. The reduction of the dataset by combining almost equal flows could make it possible to try the SVM classifier with non-linear kernels.

In our feature-vector there is only one value that is extracted from the time stamp inside the network packets. Using the time stamp is tricky because it can reflect the layout of the network. But the time stamp can also have valuable information about the reaction time during client and server communication. Could it not be possible to remove the time component that is network dependent and calculate reaction time of the client or server? By nature, a Botnet will try to hide his presence, making sure the execution of other programs is not delayed. This can have influence on the reaction time between client and server in a botnet. The reaction time can be an important feature to detect botnets. Maybe the time features introduced by Wang et al. (Wang et al. 2020) can be a starting point.

A last remark is the use of the source and destination port of the network packets. In TCP/IP communication the destination port is a fixed port where a server is listening on to receive a connection request from a client. This number can be specific for a botnet and important to use. But it is also easy to change and probably different for every types of botnet. The use of the destination port can be a problem when trying to detect new type of botnets. The destination port in a TCP/IP session is a more random number. When used for detecting botnets it can be specific for the used dataset but will be different in other situations. Therefore, it is questionable to use it. In new research it is probably better to remove them from the feature-vector to get an algorithm that performs better in detecting botnets in new environments.

7.4. Future work

During this research, new questions have surfaced. During the answering of the sub question new ones have raised.

The first question is about datasets. We already concluded that the datasets are relatively old. The question is if the network packets inside the dataset are still relevant. The programs and botnets used to create network packets are not used any more or have evolved. New datasets should be created, from recent programs, to verify that newly created algorithms also work when new data is classified. Wang et al. (Wang et al. 2020) created a more recent dataset, maybe this can be a basis for new research.

The second question is about the features used. We have used a small feature-vector of only 17 features that give perfect results. Only 11 of these features are extracted from the payload size of the network packets inside a flow. For a botnet programmer, it is probably easy to change the payload size when the botnet evolves. Are flows the best idea to detect botnets in network packets. Chowdhury (Chowdhury et al. 2017) is also questioning the flow-based method. New features that are less sensitive for change should be researched as discussed in the previous section. A second clue that network flows are maybe not enough to detect botnets is if we compare the results from Roosmalen (Roosmalen et al. 2018) and the results from this research. The results are almost equal. The neural network used by Roosmalen is using a vast neural network with all values from a network flow. The expectation is that the neural network will find all information that is important to make a classification. Because the results are not better than a random forest tree with a small feature-vector we can ask ourselves if there is any more information available inside the network flow, is a network flow enough to make a classification.

Then the used machine-learning algorithms. The way this research is setup makes it easy to try new algorithms or change the flow-based features. Because the proposed feature-vector is small, only 17 features, more algorithms can be tried. The long training and evaluation time of neural networks is hugely reduced when the input feature-vector is smaller. The network can be much smaller. Scikit-learn also has a neural network trainer that can be easily tried. Maybe a neural network is better in detecting unseen data.

This research is focusing on TCP/IP network traffic. But the used features can also be calculated when using UDP/IP network traffic. The created flows from the botshot-java program includes TCP/IP and UDP/IP data. A question could be if the Random Forest Classifier is also able to detect botnets in UDP/IP network packets.

The Random Forest Classifier is good at detecting botnets. Only when there are too many botnets in one dataset the performance drops. The question is if the Random Forest Classifier can detect one type of botnet in a large dataset. When all data is combined, and the Random Forest Classifier is trained to detect only one kind of botnet does the classifier do a better job. This can be done for all known botnets, and the set of Random Forest Classifier can work in parallel to detect all botnets. The botshot-java program can be adjusted to not only give a binary truth value but a truth value that tells what kind of botnet it is. That can be used to train multiple Random Forest Classifiers.

This research is focusing on the performance of the classifier during training and validation. The time to train and validate a classifier is important. But most time is consumed in creating flows. The process of creating flows should be optimized, so its usable on live captured data. The botshot-java

program can be easily adapted to capture live data, but the question is if it is fast enough to process the data.

The created botshot programs make it easier to continue the research on botnets. The botshot-java program is easily adaptable to use new datasets or calculate new features. Adding a new truth value should not be a problem and can be implemented fast. The botshot-python programs are easy to read, and new training methods can be easily implemented. The botshot programs are designed to be reused and adapted. Hopefully, the botshot programs can help to speed up new research on detection botnets in network traffic.

7.5. Reflection

This thesis is the result of 10 months of hard work. During this time, there were some surprises, and not everything went according to plan. That there are surprises is a part of doing research, but some could have been avoided.

The biggest surprise was the ability to find software that could help to create network flows from network packets. Although there were some programs, the usability was limited. The first couple of weeks, I tried to find a standard program but eventually decided to create my own programs (Botshot). This consumed a lot of time. In my research preparation, I had assumed the creation of flows had only taken me four weeks, but it took 12 weeks. It was consuming about half of the available programming time. During the research preparation, this should have been clear.

Another surprise that I discovered too late was a property of the SVM classifier. It becomes clear that SVM is challenging to use with large datasets. Maybe a different machine learning algorithm was a better choice.

I gained experience in the used technologies during the project. Using java and python was the first time for me in such a large project. The choice to use Java for the time-consuming part and python for the machine-learning part was working great. Both languages are easy to learn, and there are more than enough examples available on the internet to help. I hope that new researchers can use the Botshot programs to speed-up their research.

At last, I should mention the help I have got from Harald Vranken and Arjen Hommerson. The periodic discussion ware constructive and helped focusing and motivate me to continue. Without this regular feedback, it would be much harder to stick to the planning. Harald and Arjen thanks for the support I could not wish for a better team.

8. Academic references

Aizerman, M. A., È. M. Braverman, and L. I. Rozonoèr. 1964. "Theoretical Foundation of Potential Functions Method in Pattern Recognition." *Avtomat. i Telemekh.* 25 (6): 917–36.

Alauthaman, Mohammad, Nauman Aslam, Li Zhang, Rafe Alasem, and M. A. Hossain. 2018. "A P2P Botnet Detection Scheme Based on Decision Tree and Adaptive Multilayer Neural Networks." *Neural Computing and Applications* 29 (11): 991–1004. https://doi.org/10.1007/s00521-016-2564-5.

Alauthman, Mohammad, Nauman Aslam, Mouhammd Al-kasassbeh, Suleman Khan, Ahmad Al-Qerem, and Kim-Kwang [Raymond Choo]. 2020. "An Efficient Reinforcement Learning-Based Botnet Detection Approach." *Journal of Network and Computer Applications* 150: 102479. https://doi.org/10.1016/j.jnca.2019.102479.

Ali, Shawkat, and Kate A. Smith-Miles. 2006. "A Meta-Learning Approach to Automatic Kernel Selection for Support Vector Machines." *Neural Networks* 70 (1): 173–86. https://doi.org/10.1016/j.neucom.2006.03.004.

Aruoba, S. Borağan, and Jesús Fernández-Villaverde. 2014. "A Comparison of Programming Languages in Economics." Working paper 20263. Working Paper Series. National Bureau of Economic Research. https://doi.org/10.3386/w20263.

Batista, Gustavo E. A. P. A., Ronaldo C. Prati, and Maria Carolina Monard. 2004. "A Study of the Behavior of Several Methods for Balancing Machine Learning Training Data." *SIGKDD Explor. Newsl.* 6 (1): 20–29. https://doi.org/10.1145/1007730.1007735.

Biglar Beigi, E., H. Hadian Jazi, N. Stakhanova, and A. A. Ghorbani. 2014. "Towards Effective Feature Selection in Machine Learning-Based Botnet Detection Approaches." In *2014 IEEE Conference on Communications and Network Security*, 247–55. https://doi.org/10.1109/CNS.2014.6997492.

Breiman, Leo. 2001. "Random Forests." *Machine Learning* 45 (1): 5–32. https://doi.org/10.1023/A:1010933404324.

Briscoe, Erica, and Jacob Feldman. 2011. "Conceptual Complexity and the Bias/Variance Tradeoff." *Cognition* 118 (1): 2–16. https://doi.org/10.1016/j.cognition.2010.10.004.

Browne, Michael W. 2000. "Cross-Validation Methods." *Journal of Mathematical Psychology* 44 (1): 108–32. https://doi.org/10.1006/jmps.1999.1279.

Chandrashekar, Girish, and Ferat Sahin. 2014. "A Survey on Feature Selection Methods." *40th-Year Commemorative Issue* 40 (1): 16–28. https://doi.org/10.1016/j.compeleceng.2013.11.024.

Chen, Ruidong, Weina Niu, Xiaosong Zhang, Zhongliu Zhuo, and Fengmao Lv. 2017. "An Effective Conversation-Based Botnet Detection Method." Edited by Lixiang Li. *Mathematical Problems in Engineering* 2017 (April): 4934082. https://doi.org/10.1155/2017/4934082.

Chowdhury, Sudipta, Mojtaba Khanzadeh, Ravi Akula, Fangyan Zhang, Song Zhang, Hugh Medal, Mohammad Marufuzzaman, and Linkan Bian. 2017. "Botnet Detection Using Graph-Based Feature Clustering." *Journal of Big Data* 4 (1): 14. https://doi.org/10.1186/s40537-017-0074-7.

Cortes, Corinna, L. D. Jackel, and Wan-Ping Chiang. 1995. "Limits on Learning Machine Accuracy Imposed by Data Quality." In *Advances in Neural Information Processing Systems*, edited by G. Tesauro, D. Touretzky, and T. Leen, 7:239–46. MIT Press.

https://proceedings.neurips.cc/paper/1994/file/1e056d2b0ebd5c878c550da6ac5d3724-Paper.pdf.
Cortes, Corinna, and Vladimir Vapnik. 1995. "Support-Vector Networks." *Machine Learning* 20 (3): 273–97. https://doi.org/10.1007/BF00994018.
Dongarra, Jack J. 1988. "The LINPACK Benchmark: An Explanation." In *Supercomputing*, edited by E. N. Houstis, T. S. Papatheodorou, and C. D. Polychronopoulos, 456–74. Berlin, Heidelberg: Springer Berlin Heidelberg.
Dongarra, Jack J., Piotr Luszczek, and Antoine Petitet. 2003. "The LINPACK Benchmark: Past, Present and Future." *Concurrency and Computation: Practice and Experience* 15 (9): 803–20. https://doi.org/10.1002/cpe.728.
Drašar, Martin, Martin Vizváry, and Jan Vykopal. 2014. "Similarity as a Central Approach to Flow-Based Anomaly Detection." *International Journal of Network Management* 24 (4): 318–36. https://doi.org/10.1002/nem.1867.
Fawcett, Tom. 2006. "An Introduction to ROC Analysis." *ROC Analysis in Pattern Recognition* 27 (8): 861–74. https://doi.org/10.1016/j.patrec.2005.10.010.
Fernandez-Delgado, Manuel, E. Cernadas, S. Barro, and Dinani Amorim. 2014. "Do We Need Hundreds of Classifiers to Solve Real World Classification Problems?" *Journal of Machine Learning Research* 15 (October): 3133–81.
Friedman, Jerome H. 2001. "Greedy Function Approximation: A Gradient Boosting Machine." *The Annals of Statistics* 29 (5): 1189–1232.
García, S., M. Grill, J. Stiborek, and A. Zunino. 2014. "An Empirical Comparison of Botnet Detection Methods." *Computers & Security* 45 (September): 100–123. https://doi.org/10.1016/j.cose.2014.05.011.
Guyon, Isabelle, and Andre Elisseeff. 2003. "An Introduction to Variable and Feature Selection." *J. Mach. Learn. Res.* 3: 1157–82.
Ho, Tin Kam. 1995. "Random Decision Forests." In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, 1:278–82 vol.1. IEEE. https://doi.org/10.1109/ICDAR.1995.598994.
Khattak, Sheharbano, Naurin Ramay, Kamran Khan, Affan Syed, and Syed Khayam. 2014. "A Taxonomy of Botnet Behavior, Detection, and Defense." *Communications Surveys & Tutorials, IEEE* 16 (January): 898–924. https://doi.org/10.1109/SURV.2013.091213.00134.
Klaveren, Alexander van. 2019. "Understanding the Inner Workings of a Deep Neural Network." Open University.
Koroniotis, Nickolaos, Nour Moustafa, Elena Sitnikova, and Benjamin Turnbull. 2019. "Towards the Development of Realistic Botnet Dataset in the Internet of Things for Network Forensic Analytics: Bot-IoT Dataset." *Future Generation Computer Systems* 100: 779–96. https://doi.org/10.1016/j.future.2019.05.041.
Krawczyk, Bartosz. 2016. "Learning from Imbalanced Data: Open Challenges and Future Directions." *Progress in Artificial Intelligence* 5 (4): 221–32. https://doi.org/10.1007/s13748-016-0094-0.
Lin, Kuan-Cheng, Sih-Yang Chen, and Jason C. Hung. 2014. "Botnet Detection Using Support Vector Machines with Artificial Fish Swarm Algorithm." Edited by Young-Sik Jeong. *Journal of Applied Mathematics* 2014 (April): 986428. https://doi.org/10.1155/2014/986428.
Louppe, Gilles. 2015. "Understanding Random Forests: From Theory to Practice." Cornwell University. https://arxiv.org/abs/1407.7502.
Matthews, B.W. 1975. "Comparison of the Predicted and Observed Secondary Structure of T4 Phage Lysozyme." *Biochimica et Biophysica Acta (BBA) -*

Protein Structure 405 (2): 442–51. https://doi.org/10.1016/0005-2795(75)90109-9.

Moons, Karel G M, Douglas G Altman, Yvonne Vergouwe, and Patrick Royston. 2009. "Prognosis and Prognostic Research: Application and Impact of Prognostic Models in Clinical Practice." *BMJ* 338. https://doi.org/10.1136/bmj.b606.

Nadji, Yacin, Manos Antonakakis, Roberto Perdisci, David Dagon, and Wenke Lee. 2013. "Beheading Hydras: Performing Effective Botnet Takedowns." In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 121–32. CCS '13. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2508859.2516749.

Pektaş, Abdurrahman, and Tankut Acarman. 2019. "Deep Learning to Detect Botnet via Network Flow Summaries." *Neural Computing and Applications* 31 (11): 8021–33. https://doi.org/10.1007/s00521-018-3595-x.

Poon, Tho. 2018. "Botnet Detection Using Recurrent Neural Networks." Open University.

Rahbarinia, Babak, Roberto Perdisci, Andrea Lanzi, and Kang Li. 2014. "PeerRush: Mining for Unwanted P2P Traffic." *Journal of Information Security and Applications* 19 (3): 194–208. https://doi.org/10.1016/j.jisa.2014.03.002.

rfc, 793. 1981. "Transmission Control Protocol." Internet Engineering Task Force. https://rfc-editor.org/rfc/rfc793.txt.

Roosmalen, Jos van. 2017. "The Feasibility of Deep Learning Approaches for P2P-Botnet Detection." Open University.

Roosmalen, Jos, Harald Vranken, and Marko Eekelen. 2018. *Applying Deep Learning on Packet Flows for Botnet Detection*. Association for Computing Machinery. https://doi.org/10.1145/3167132.3167306.

Saad, S, I Traore, A Ghorbani, B Sayed, D Zhao, Wei Lu, Felix, and P Hakimian. 2011. "Detecting P2P Botnets through Network Behavior Analysis and Machine Learning." In *2011 Ninth Annual International Conference on Privacy, Security and Trust*, 174–80. https://doi.org/10.1109/PST.2011.5971980.

Sokolova, Marina, and Guy Lapalme. 2009. "A Systematic Analysis of Performance Measures for Classification Tasks." *Information Processing & Management* 45 (4): 427–37. https://doi.org/10.1016/j.ipm.2009.03.002.

Vasques, Thiago Lara, Pedro Moura, and Aníbal de Almeida. 2019. "A Review on Energy Efficiency and Demand Response with Focus on Small and Medium Data Centers." *Energy Efficiency* 12 (5): 1399–1428. https://doi.org/10.1007/s12053-018-9753-2.

Wang, Wei, Yaoyao Shang, Yongzhong He, Yidong Li, and Jiqiang Liu. 2020. "BotMark: Automated Botnet Detection with Hybrid Analysis of Flow-Based and Graph-Based Traffic Behaviors." *Information Sciences* 511: 284–96. https://doi.org/10.1016/j.ins.2019.09.024.

Widanapathirana, Chathuranga, Ahmet Sekercioglu, Milosh Ivanovich, Paul Fitzpatrick, and Jonathan Li. 2012. "Automated Inference System for End-To-End Diagnosis of Network Performance Issues in Client-Terminal Devices." *International Journal of Computer Networks & Communications (IJCNC)* 4: 37–56.

Wong, Tzu-Tsung. 2015. "Performance Evaluation of Classification Algorithms by K-Fold and Leave-One-out Cross Validation." *Pattern Recognition* 48 (9): 2839–46. https://doi.org/10.1016/j.patcog.2015.03.009.

Zeidanloo, Hossein Rouhani, M. Shooshtari, P. V. Amoli, M. Safari, and M. Zamani. 2010. "A Taxonomy of Botnet Detection Techniques." *2010 3rd International Conference on Computer Science and Information Technology* 2: 158–62.

Zhao, David, Issa Traore, Bassam Sayed, Wei Lu, Sherif Saad, Ali Ghorbani, and Dan Garant. 2013. "Botnet Detection Based on Traffic Behavior Analysis and Flow Intervals." *27th IFIP International Information Security Conference* 39 (November): 2–16. https://doi.org/10.1016/j.cose.2013.04.007.

Zou, James, Mikael Huss, Abubakar Abid, Pejman Mohammadi, Ali Torkamani, and Amalio Telenti. 2019. "A Primer on Deep Learning in Genomics." *Nature Genetics* 51 (1): 12–18. https://doi.org/10.1038/s41588-018-0295-5.

9. Non-academic references

Fedorov, Gennady, Shaojan Zhu, and Abhinav Singh. 2020. *Intel® Math Kernel Library (Intel® MKL) Benchmarks Suite.* 20 08. Accessed 10 20, 2020. https://software.intel.com/content/www/us/en/develop/articles/intel-mkl-benchmarks-suite.html.

Gorham, Matt. 2019. *2019 Internet Crime Report.* Federal Bureau of Investigation. Accessed 12 21, 2020. https://pdf.ic3.gov/2019_IC3Report.pdf.

Huilgol, Purva. 2020. *Bias and Variance in Machine Learning – A Fantastic Guide for Beginners!* Accessed Nov 2020. https://www.analyticsvidhya.com/blog/2020/08/bias-and-variance-tradeoff-machine-learning/.

Kozierok, Charles . 2005. *The TCP/IP guide: A comprehensive, illustrated internet protocols reference.* No Starch Press. http://www.tcpipguide.com.

Oracle. 2020. *Java Language and Virtual Machine Specifications.* 9. Accessed 12 22, 2020. https://docs.oracle.com/javase/specs/.

Python. 2020. *The Python Language Reference.* 17 12. Accessed 12 22, 2020. https://docs.python.org/3/reference/.

rfc 793. 1981. "Transmission Control Protocol." September. https://rfc-editor.org/rfc/rfc793.txt.

scikit-learn. 2020. *scikit-learn.* 23 10. Accessed 10 23, 2020. www.scikit-learn.org.

Wireshark. 2020. *Wireshark.* 13 11. Accessed 2020. https://www.wireshark.org/.

www.jupyter.org. 2020. *jupyter.org.* 21 10. Accessed 20 26, 2020. www.jupyter.org.

Appendix A

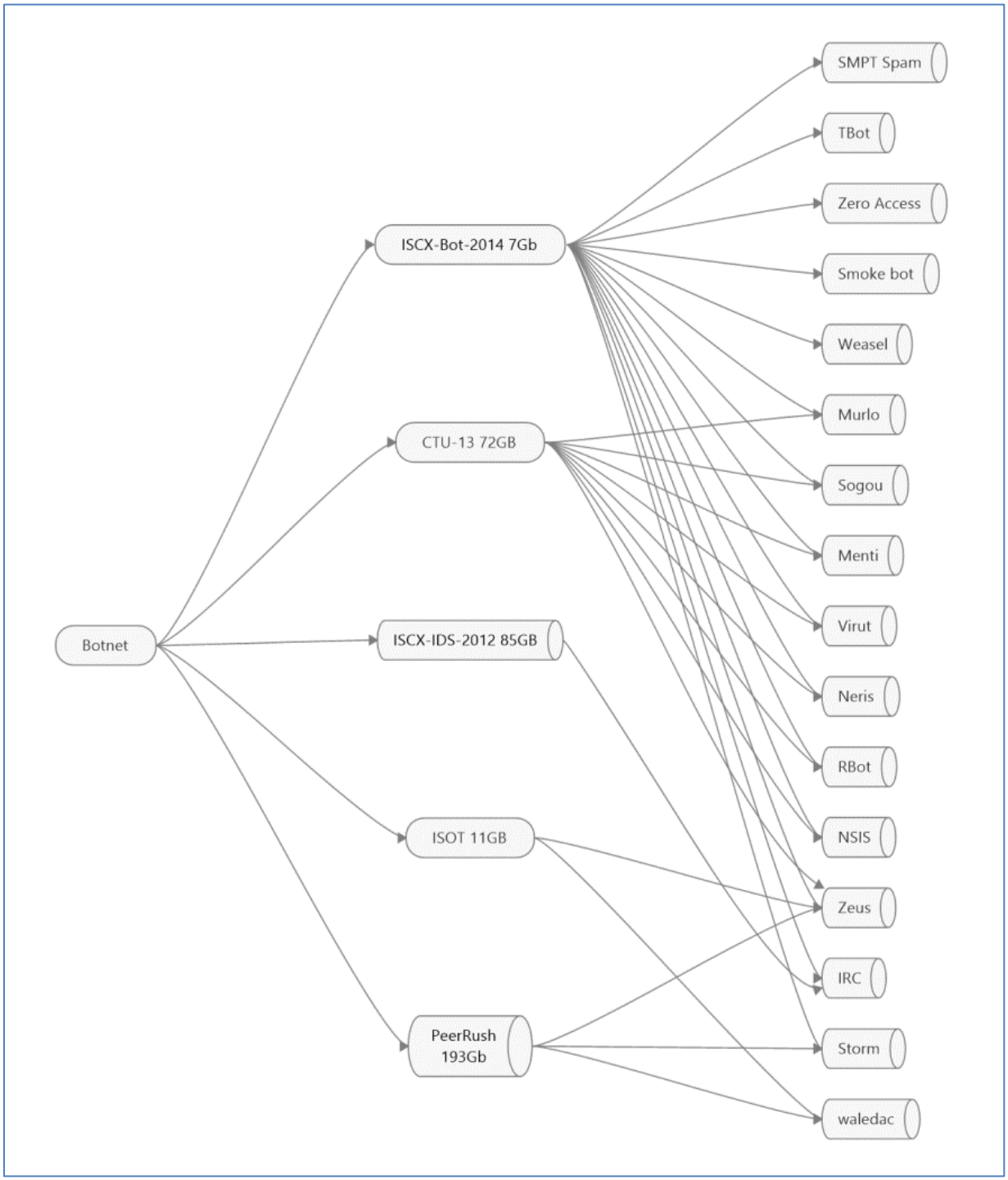


Figure 21: Included botnets

Appendix B

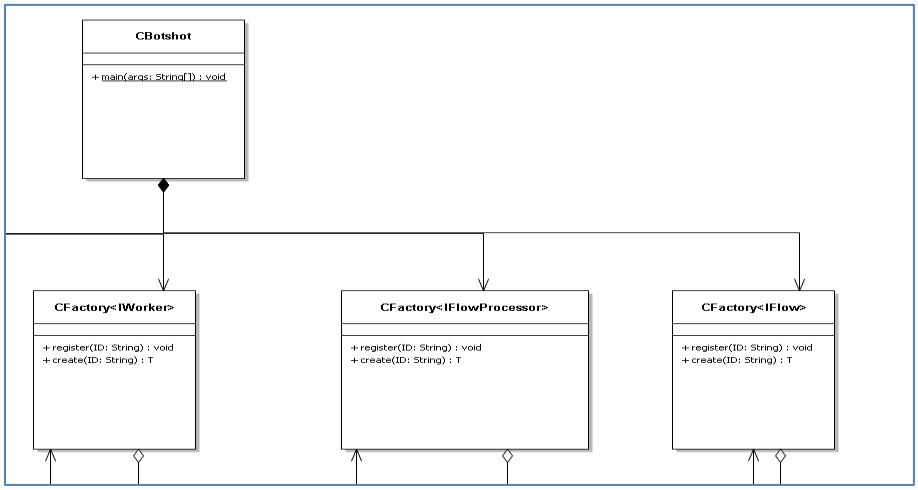


Figure 22: Botshot Java Factory classes

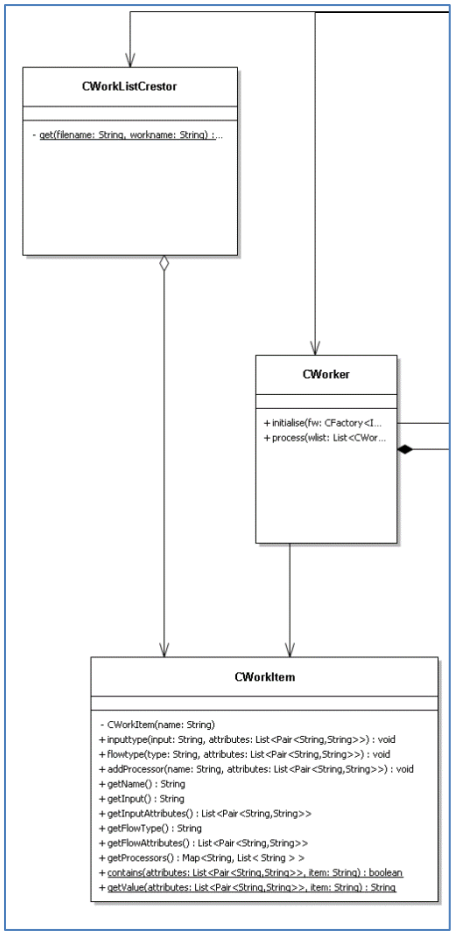


Figure 24: Botshot Java configuration

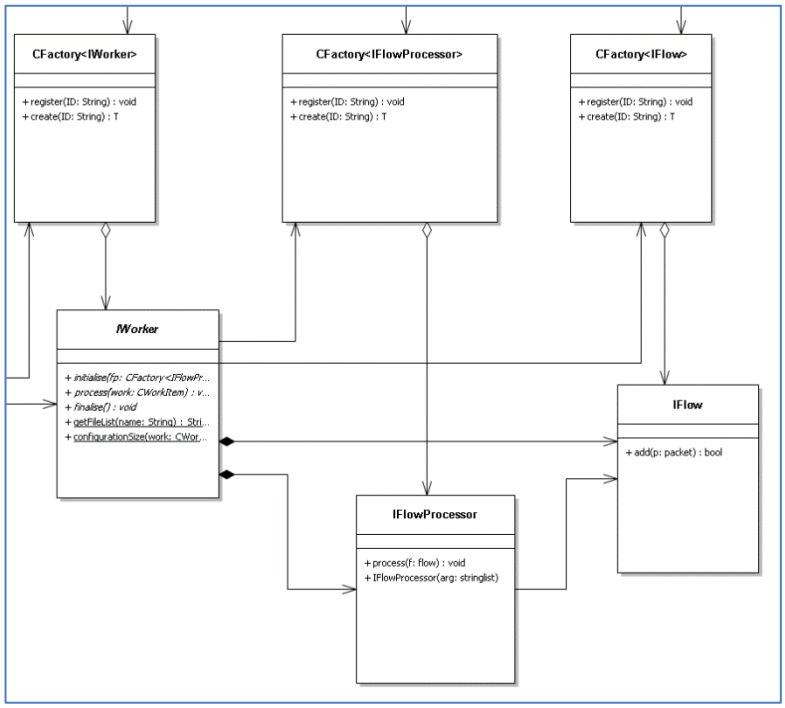


Figure 23 : Botshot Java processing

Appendix C

IWorker

We implemented two IWorker classes. They are configured in the main function for use.

```
/* create a factory class that creates the workers  */
CFactory<IWorker> fwfactory = new CFactory<>();
/* register the worker classes with the worker factory */
fwfactory.register("pcap",CWorkerPCAP.class);
fwfactory.register("csv",CWorkerCSV.class);
```

CWorkerPCAP	**Uses a PCAP file as an input and creates a flow from the network packets**
Inputs	
file	A list of files separated with ';'. The files names can have wildcard to select multiple files at ones.

CWorkerCSV	**Uses a CSV file as an input to read the flow**
Inputs	
file	A list of files separated with ';'. The files names can have wildcard to select multiple files at ones.
minPacCnt	Minimum number of network packets before it is a flow.

IFlowProcessor

All functions derived from IFlowProcessor are used to add information to the flow or export the flow. This can be the truth value or the output of a comma separated file for further processing. The different classes are registered to a factory in the main function.

```
/* create a factory class that creates the processors for flows */
CFactory<IFlowProcessor> fpfactory = new CFactory<>();
/* register the processing classes with the flow processor factory */
fpfactory.register("Exporter",CFlowProcessor_Exporter.class);
fpfactory.register("Statistics",CFlowProcessor_Statistics.class);
fpfactory.register("Generic",CFlowProcessor_Generic.class);
fpfactory.register("GenericIP",CFlowProcessor_IP.class);
fpfactory.register("CISCXIDS2012",CFlowProcessor_CISCXIDS2012.class);
```

CFlowProcessor_Exporter	**Output a flow in a comma separated file**
Inputs	
file	Output file name

CFlowProcessor_Statistics	**Output a file with general statistics about all flows**
Inputs	
file	Output file name

CFlowProcessor_Generic	**Add a constant truth value to a flow**
Inputs	
value	Truth value. True / False

CFlowProcessor_IP	**Select the Botnet flow according to the IP address and/or MAC address**
Inputs	
ip	The IP number and/or MAC of a flow from a botnet. The format is 172.16.2.11[BB:BB:BB:BB:BB:BB]

CFlowProcessor_CISCXIDS2012	**Add truth information to the flows of the ISCX IDS 2012 dataset**
Inputs	
file	Input file with truth information

IFlow

IFow is the base class for holding the flow information. There are tree classes created as an implementation for the flow class. The class CFlow_Base is a basic implementation and not used directly. Both classes CFlow_Simple and CFlow_Sets are derived from the CFlow_Base class.

```
/* create a factory class that creates the flow containers */
CFactory<IFlow> ffactory = new CFactory<>();
/* register the flow class with the flow factory */
ffactory.register("Basic",CFlow_Base.class);
ffactory.register("Simple",CFlow_Simple.class);
ffactory.register("Sets",CFlow_Sets.class);
```

CFlow_Simple	**Holder of flows and used with *CWorkerPCAP* class**
Inputs	
Timeout	When does a flow ends after the last network packet
Skipport	A comma separated list of ports that should be skipped. This improves the speed of processing when there is no need to create flows for uninteresting ports. Especially DNS entries can be skipped this way.

CFlow_Sets	**Holder of flows and used with *CWorkerCSV* class**
Inputs	

Appendix D
Botshot Balance the dataset

Configuration variable

```
FileSrc = '../flows/validate.flw'
fieldnames =
['target','type','duration','finished','first','src','dst','total','hop','push','u
rgent','sync','rst']

FileDstUnder = '../flows/train_balanced_under.flw'
FileDstOver = '../flows/train_balanced_over.flw'
```

First open de CSV file

```
import numpy as np
import pandas as pd

df_train = pd.read_csv(FileSrc)

target_count = df_train.target.value_counts()
print('Class 0:', target_count[0])
print('Class 1:', target_count[1])
print('Proportion:', round(target_count[0] / target_count[1], 2), ': 1')

target_count.plot(kind='bar', title='Count (target)');
```

re-sample the dataset

```
from sklearn.utils import resample

count_class_0, count_class_1 = df_train.target.value_counts()

# Divide by class
df_class_0 = df_train[df_train['target'] == 0]
df_class_1 = df_train[df_train['target'] == 1]

#under-sample
df_class_0_under = resample(df_class_0,
                            replace=False,    # sample without replacement
                            n_samples=len(df_class_1.index),     # to match
minority class
                            random_state=123) # reproducible results

df_class_0_under = df_class_0.sample(count_class_1)
df_test_under = pd.concat([df_class_0_under, df_class_1], axis=0)
target_count_under = df_test_under.target.value_counts()

print('Random under-sampling:')
print('Class 0:', target_count_under[0])
print('Class 1:', target_count_under[1])

df_test_under.target.value_counts().plot(kind='bar', title='Count (target)');

#over-sample
df_class_1_over = resample(df_class_1,
                           replace=True,     # sample with replacement
                           n_samples=len(df_class_0.index),    # to match majority
```

```
class
                            random_state=123) # reproducible results
df_test_over = pd.concat([df_class_1_over, df_class_0], axis=0)
target_count_over = df_test_over.target.value_counts()

print('Random over-sampling:')
print('Class 0:', target_count_over[0])
print('Class 1:', target_count_over[1])

df_test_over.target.value_counts().plot(kind='bar', title='Count (target)');
```

Save the balanced file in CSV format

```
df_test_under.to_csv(FileDstUnder, index = False)
df_test_over.to_csv(FileDstOver, index = False)
```

Botshot Trainer

Configuration variable

```
FileSrc = '../flows/train_balanced_under.flw'
Type    = 'RFC' #'SVM''GBC' 'RFC'

if (Type == 'SVM'):
    FileModel = './models/svm.joblib'
    ROCFile   = './models/svm.png'
    FeatFile  = './models/svm.txt'
    LogFile   = './models/svm.log'

if (Type == 'RFC'):
    FileModel = './models/rfc.joblib'
    ROCFile   = './models/rfc.png'
    FeatFile  = './models/rfc.txt'
    LogFile   = './models/rfc.log'

if (Type == 'GBC'):
    FileModel = './models/gbc.joblib'
    ROCFile   = './models/gbc.png'
    FeatFile  = './models/gbc.txt'
    LogFile   = './models/gbc.log'

print('Training using : ',Type)
```

Initialize the logger

```
import logging
import sys

logger = logging.getLogger('botshot.training')
logger.addHandler(logging.NullHandler())

file_log_handler = logging.FileHandler(LogFile, mode='w')
logger.addHandler(file_log_handler)

stderr_log_handler = logging.StreamHandler(sys.stdout)

logger.addHandler(stderr_log_handler)

# nice output format
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s -
%(message)s')
file_log_handler.setFormatter(formatter)
#stderr_log_handler.setFormatter(formatter)
logger.setLevel('DEBUG')

logger.info('Training using : %s',Type)
```

Open the CSV file

```
import numpy as np
import pandas as pd

df_train = pd.read_csv(FileSrc)

target_count = df_train.target.value_counts()
logger.info('Class 0: %d', target_count[0])
```

```
logger.info('Class 1: %d', target_count[1])
logger.info('Proportion: %.2f , %s', round(target_count[0] / target_count[1], 2),
': 1')
```

Split the set in two

```
from sklearn.model_selection import train_test_split

# remove these features from the list
df_train_tmp = df_train.drop(columns=['type','push','sync','urg','hop'])

x = df_train_tmp.iloc[:,1:]
y = df_train_tmp['target']

print(x.shape)
print(y.shape)

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.25,
random_state=42)

logger.info('y train size %s',str(y_train.shape))
logger.info('x train size %s',str(x_train.shape))
logger.info('y test size  %s',str(y_test.shape))
logger.info('x test size  %s',str(x_test.shape))
```

Make a pipeline and fit the model

```
from sklearn.svm import LinearSVC
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import Normalizer
import timeit

if (Type == 'SVM'):
    classifier = make_pipeline(Normalizer(),LinearSVC(dual=False,
                                                       random_state=0,
                                                       tol=1e-5,
                                                       verbose=1))
    #classifier = make_pipeline(Normalizer(),SVC(kernel='linear',
    #                                            cache_size=4000,
    #                                            verbose=1))

if (Type == 'RFC'):
    classifier = make_pipeline(Normalizer(),RandomForestClassifier

              (n_estimators=40,
               n_jobs=-1,
               max_depth=30,
               min_samples_split=2,
               random_state=42,
               verbose=1))

if (Type == 'GBC'):
    classifier = make_pipeline(Normalizer(),GradientBoostingClassifier

              (n_estimators=100,
               learning_rate=0.1,
```

```
                max_depth=4,
                random_state=0,
                verbose=1))
starttime = timeit.default_timer()

classifier.fit(x_train, y_train)

logger.info('time to train : %.2f',round(timeit.default_timer() - starttime,2))
```

Get the feature Importance

```
from sklearn.inspection import permutation_importance
import matplotlib.pyplot as plt
import os

if (Type == 'RFC' or Type == 'GBC'):
    feature_importances = pd.DataFrame(classifier[1].feature_importances_,
                                       index = x_train.columns,

columns=['importance']).sort_values('importance',ascending=False)

else:
    feature_importances = pd.DataFrame(np.swapaxes(classifier[1].coef_,0,1),
                                       index = x_train.columns,

columns=['importance']).sort_values('importance',ascending=False)

ax = feature_importances.plot.bar(figsize=(20,5),log=True)
ax.set(title="Feature importance")

plt.show()
feature_importances.to_csv(FeatFile)
filename = os.path.splitext(FeatFile)[0] + '_imp.png'
print (filename)
ax.get_figure().savefig(os.path.splitext(FeatFile)[0] + '_imp.png')
```

Calculate the ACC and MCC

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import matthews_corrcoef

starttime = timeit.default_timer()
y_pred = classifier.predict(x_test)
logger.info('time to predict : %.2f',round(timeit.default_timer() - starttime,2))

logger.info('accuracy : %.2f', round(accuracy_score(y_test,y_pred)*100,2))
logger.info('MCC      : %.2f', round(matthews_corrcoef(y_test, y_pred)*100,2))
```

Plot a ROC curve

```
import matplotlib.pyplot as plt
from sklearn.metrics import auc
from sklearn.metrics import plot_roc_curve

fig, ax = plt.subplots()

viz = plot_roc_curve(classifier, x_test, y_test,name=Type,alpha=0.3, lw=1, ax=ax)

ax.plot([0, 1], [0, 1], linestyle='--', lw=1, color='r',label='Chance', alpha=.8)
```

```
ax.set(xlim=[-0.05, 1.05], ylim=[-0.05, 1.05],title="Receiver operating
characteristic")
ax.legend(loc="lower right")
plt.show()
fig.savefig(ROCFile,bbox_inches='tight')
print(ROCFile)
```

Dump the model

```
from joblib import dump, load

dump(classifier, FileModel)
logging.shutdown()
```

Botshot Optimizer

Configuration variable

```
FileSrc = '../flows/flow_16/train_balanced_under.flw'
Type    = 'RFC' #'SVM''GBC' 'RFC'

if (Type == 'SVM'):
    rmFile   = './models/svm_o.png'
    FeatFile  = './models/svm_o.txt'
    LogFile   = './models/svm_o.log'

if (Type == 'RFC'):
    rmFile   = './models/rfc_o.png'
    FeatFile  = './models/rfc_o.txt'
    LogFile   = './models/rfc_o.log'

if (Type == 'GBC'):
    rmFile   = './models/gbc_o.png'
    FeatFile  = './models/gbc_o.txt'
    LogFile   = './models/gbc_o.log'

print('Training using : ',Type)
```

Initialize the logger

```
import logging
import sys

logger = logging.getLogger('botshot.training')
logger.addHandler(logging.NullHandler())

file_log_handler = logging.FileHandler(LogFile, mode='w')
logger.addHandler(file_log_handler)

stderr_log_handler = logging.StreamHandler(sys.stdout)

logger.addHandler(stderr_log_handler)

# nice output format
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s -
%(message)s')
file_log_handler.setFormatter(formatter)
#stderr_log_handler.setFormatter(formatter)
logger.setLevel('DEBUG')

logger.info('Training using : %s',Type)
```

Open the CSV file

```
import numpy as np
import pandas as pd

df_train = pd.read_csv(FileSrc)

target_count = df_train.target.value_counts()
logger.info('Class 0: %d', target_count[0])
logger.info('Class 1: %d', target_count[1])
logger.info('Proportion: %.2f , %s', round(target_count[0] / target_count[1], 2),
': 1')
```

Split the set in two

```
from sklearn.model_selection import train_test_split

# remove these features from the list
df_train_dropped = df_train.drop(columns=['type','push','sync','urg','hop'])

x = df_train_dropped.iloc[:,1:]
y = df_train_dropped['target']

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.25,
random_state=42)

logger.info('y train size %s',str(y_train.shape))
logger.info('x train size %s',str(x_train.shape))
logger.info('y test size  %s',str(y_test.shape))
logger.info('x test size  %s',str(x_test.shape))
```

Train a model and remove the least important feature

```
from sklearn.svm import LinearSVC
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import Normalizer
import timeit
from sklearn.metrics import accuracy_score
from sklearn.metrics import matthews_corrcoef

df_results = pd.DataFrame (columns = ['acc','mcc'])

for i in range(len(x_train.columns)) :
    print(i)

    if (Type == 'SVM'):
        classifier = make_pipeline(Normalizer(),LinearSVC

                    (dual=False,
                     random_state=0,
                     tol=1e-5,
                     verbose=1))
    if (Type == 'RFC'):
        classifier = make_pipeline(Normalizer(),RandomForestClassifier

                   (n_estimators=40,
                    n_jobs=-1,
                    max_depth=30,
                    min_samples_split=2,
                    random_state=42,
                    verbose=0))

    if (Type == 'GBC'):
        classifier = make_pipeline(Normalizer(),GradientBoostingClassifier

                   (n_estimators=400,
                    learning_rate=0.1,
                    max_depth=4,
                    random_state=0,
```

```
                    verbose=1))


    classifier.fit(x_train, y_train)
    y_pred = classifier.predict(x_test)

    acc = round(accuracy_score(y_test,y_pred)*100,2)
    mcc = round(matthews_corrcoef(y_test, y_pred)*100,2)

    print ('accuracy = ' + str(acc) + ' MCC = ' + str(mcc))

    if (Type == 'RFC' or Type == 'GBC'):
        feature_importances = pd.DataFrame(classifier[1].feature_importances_,
                                           index = x_train.columns,

columns=['importance']).sort_values('importance',ascending=False)
    else:
        feature_importances = pd.DataFrame(np.swapaxes(classifier[1].coef_,0,1),
                                           index = x_train.columns,

columns=['importance']).sort_values('importance',ascending=False)

    df_results = df_results.append(pd.Series({'acc': acc, 'mcc': mcc},
name=feature_importances.index[-1]))
    print (feature_importances)

    x_train = x_train.drop(columns=feature_importances.index[-1])
    x_test = x_test.drop(columns=feature_importances.index[-1])

    print('remove feature ' + feature_importances.index[-1] + ' ' +
str(x_train.shape))

df_results.to_csv("./models/optimize.csv")
```

Plot the feature list

```
import matplotlib.pyplot as plt

df_revert = df_results

fig, ax = plt.subplots(figsize=(20, 5))
for (columnName, columnData) in df_revert.iteritems():
    ax.scatter(df_revert.index,columnData,label=columnName)


for index, row in df_revert.iterrows():
    if (row['acc'] < row['mcc']) :
        off = row['acc']
    else:
        off = row['mcc']
    if (off < 20):
        if (row['acc'] > row['mcc']) :
            off = row['acc'] +30
        else:
            off = row['mcc'] +30

    ax.text(index,off-10,row['acc'])
    ax.text(index,off-20,row['mcc'])
```

```
ax.set(title="Feature removal")
ax.legend(loc='upper right')

plt.ylim(0,100)
plt.xticks(rotation=90)
plt.show()

fig.savefig(rmFile,bbox_inches='tight')
```

Botshot Validate

Configuration variable

```
FileSrc = '../flows/train_balanced_over_val.flw'
Type    = 'GBC' #'SVM''GBC' 'RFC'

if (Type == 'SVM'):
    FileModel = './models/svm.joblib'
    ROCFile   = './models/svm_V.png'
    FeatFile  = './models/svm_V.txt'
    LogFile   = './models/svm_V.log'

if (Type == 'RFC'):
    FileModel = './models/rfc.joblib'
    ROCFile   = './models/rfc_V.png'
    FeatFile  = './models/rfc_V.txt'
    LogFile   = './models/rfc_V.log'

if (Type == 'GBC'):
    FileModel = './models/gbc.joblib'
    ROCFile   = './models/gbc_V.png'
    FeatFile  = './models/gbc_V.txt'
    LogFile   = './models/gbc_V.log'

print('validate using : ',Type)
```

Initialize the logger

```
import logging
import sys

logger = logging.getLogger('botshot.validate')
logger.addHandler(logging.NullHandler())

file_log_handler = logging.FileHandler(LogFile, mode='w')
logger.addHandler(file_log_handler)

stderr_log_handler = logging.StreamHandler(sys.stdout)

logger.addHandler(stderr_log_handler)

# nice output format
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s -
%(message)s')
file_log_handler.setFormatter(formatter)
#stderr_log_handler.setFormatter(formatter)
logger.setLevel('DEBUG')

logger.info('Training using : %s',Type)
```

Open the CSV file

```
import numpy as np
import pandas as pd

# remove these features from the list

df_validate = pd.read_csv(FileSrc)
df_validate = df_validate.drop(columns=['type','push','sync','urg','hop'])
```

```
target_count = df_validate.target.value_counts()
logger.info('Class 0: %d', target_count[0])
logger.info('Class 1: %d', target_count[1])
logger.info('Proportion: %.2f , %s', round(target_count[0] / target_count[1], 2),
': 1')
```

Predict the new values

```
from joblib import dump, load
classifier = load(FileModel)

x_test = df_validate.iloc[:,1:]
y_test = df_validate['target']

from sklearn.metrics import accuracy_score
from sklearn.metrics import matthews_corrcoef
import timeit

starttime = timeit.default_timer()
y_pred = classifier.predict(x_test)
logger.info('time to predict : %.2f',round(timeit.default_timer() - starttime,2))

logger.info('accuracy : %.2f', round(accuracy_score(y_test,y_pred)*100,2))
logger.info('MCC      : %.2f', round(matthews_corrcoef(y_test, y_pred)*100,2))
```